Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Control and Computer Engineering (37$^{\text{th}}$ cycle)

Machine Learning Methods for Hardware-Based Malware Detection

Cristiano Pegoraro Chenet

* * * * *

Supervisors
Prof. Stefano Di Carlo, Supervisor
Prof. Alessandro Savino, Co-supervisor

Politecnico di Torino
April 30, 2025

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see `www.creativecommons.org`. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . .

Cristiano Pegoraro Chenet

Turin, April 30, 2025

Summary

Cyber insecurity is among the most significant global risks, alongside major threats such as climate change and involuntary migration. Malware, short for malicious software, is the primary vector for cybercrimes. It includes any code modification within a software system aimed at causing harm or disrupting the system's intended function. Malware attacks cover spying, intrusive ads, email abuse, system damage, ransom demands, data release, slowdown, browser manipulation, and unauthorized access to sensitive information. The latest attempt to boost malware detection is a hardware-based approach, also called Hardware-Based Malware Detection (HMD). It involves dynamically analyzing architecture and microarchitecture events in a Central Processing Unit (CPU) using Machine Learning (ML) algorithms to distinguish between benign applications and malware. This thesis investigated this approach in depth, focusing on how the core component - the ML methods – should be employed to optimize performance and efficiency. A survey on the field deepens knowledge about malware, its properties and classification; discusses and compares the main available approaches for malware detection; identifies and discusses key features of HMD approaches, such as the CPUs' Performance Monitoring Units (PMUs), starting point that enables the detection by hardware; and elaborates a landscape of recent papers in HMD, assessing proposed solutions in terms of performance and efficiency. Two case studies employ a simulation environment and an anomaly HMD framework to investigate the approach performance capabilities in detecting the challenging zero-day malware, a threat that exploits previously unknown vulnerabilities. These studies implement the detection for the emerging Reduced Instruction Set Computer V (RISC-V) to disseminate the hardware-based approaches to the community surrounding this platform. The findings presented in this thesis help the malware detection field. Vulnerability exploitation, propagation method, and concealment strategy, together Common Vulnerabilities Exposures (CVE) and Common Weakness Enumeration (CWE) lists, are straightforward resources to deal with malware. The advantages of HMD are the ability for runtime, zero-day, and stealthy malware detection, resilience against subverting the protection, low-performance overhead, and reduced detection cost. ML methods to improve performance in the field were discussed, such as ensemble learning, specialization, adaptive detection, and time series. Moreover, the case

studies showed the ability of anomaly HMD to detect zero-day threats, identifying some points to overcome the poor detection: classifiers should be tailored to specific applications, the randomness/unpredictability of protected applications is an obstacle, detection algorithms should match the application, and an optimal number of ML features (Hardware Performance Counters (HPCs)) helps tune performance. Finally, HMD is a promising solution, especially if combined with software detection, generating effective and lightweight detectors.

Acknowledgements

As I reach the end of this PhD journey, I would like to express my deepest and sincere gratitude to Professors Stefano Di Carlo and Alessandro Savino. They offered me a unique opportunity that has the power to change lives. I am grateful to Stefano for his constant support, availability, and understanding, particularly when I became a father during this journey. And to Alessandro, for his understanding and for closely guiding me through the most technically challenging phases.

I would also like to acknowledge the entire team of the Vitamin-V project, funded by the Horizon Europe program and coordinated by Professor Ramon Canal, for the valuable opportunities and enriching exchanges they provided. A heartfelt thank you as well to the SMILIES research group and others from Laboratory 6, who warmly welcomed me to Italy and continued to share the everyday moments, challenges, and victories of the PhD path. Special thanks also to Davide Bruno and Zhang Ziteng, with whom I collaborated at least during their Master's thesis.

And last but certainly not least, my deepest thanks to my beloved Fabiana and our little Davi. Fabiana encouraged me at the beginning of this PhD, and Davi, despite his young age, did many sacrifices that made the completion of this PhD possible.

List of Acronyms

AES Advanced Encryption Standard. 52, 53, 55, 56, 58, 59, 61, 74

AI Artificial Intelligence. 33

ALU Arithmetic Logic Unit. 29

ANN Artificial Neural Network. 36, 41, 60

API Application Programming Interface. 51, 67

AUC Area Under the Curve. 19, 46, 54, 75

BBL Berkeley Bootloader. 71, 72

BSD Berkeley Source Distribution. 65

CISC Complex Instruction Set Computer. 27

CLINT Core Local Interrupt Controller. 71

CPU Central Processing Unit. iii, xii, xiv, 2, 22, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 40, 52, 64, 66, 67, 70, 71, 72, 80

CVE Common Vulnerabilities Exposures. iii, 10, 13, 17, 80

CWE Common Weakness Enumeration. iii, 13, 80

DL Deep Learning. 18

DMA Direct Memory Access. 67

DNN Deep Neural Network. 24

DoS Denial of Service. 6, 8, 9, 10, 11, 13, 74

DSP Digital Signal Processing. 40

ENISA European Union Agency for Cybersecurity. 11

FCN Fully Convolutional Neural Network. 43

FE Feature Extraction. 21, 34, 35

FN False Negative. 19

FP False Positive. 19

FPR False Positive Rate. 19

FS Feature Selection. 34, 35, 36

GPU Graphics Processing Unit. 67

HLS High-Level Synthesis. 40

HMD Hardware-Based Malware Detection. iii, iv, xii, xiv, xv, 2, 3, 4, 13, 22, 24, 26, 27, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 48, 51, 54, 55, 60, 61, 62, 68, 71, 75, 80, 81

HPC Hardware Performance Counter. iv, xii, xiii, 22, 24, 25, 27, 29, 34, 35, 36, 37, 39, 40, 41, 42, 48, 50, 51, 52, 53, 54, 55, 57, 59, 60, 61, 62, 68, 71, 72, 75, 76, 77, 78, 81

IoT Internet of Things. 8, 64

IP Internet Protocol. 74

ISA Instruction Set Architecture. xii, 3, 26, 27, 33, 34, 67

JOP Jump-Oriented Programming. 13

KNN K-Nearest Neighbors. 36

KVM Kernel Virtual Machine. 67

LOF Local Outlier Factor. 47, 52, 55, 58, 59, 60, 72, 78

LSTM Long Short-Term Memory. 43

LZ78 Lempel-Ziv Data Compression. 47, 48

MIPS Microprocessor without Interlocked Pipelined Stages. xiv, 29, 30, 67

ML Machine Learning. iii, iv, xii, 2, 3, 4, 17, 18, 21, 22, 24, 27, 36, 37, 40, 41, 42, 43, 45, 46, 48, 49, 60, 61, 80, 81

MLP Multilayer Perceptron. 41, 42

MMU Memory Management Unit. 71

MRD Malware Runtime Detection. 81

MSR Model-specific Register. 33

NIST National Institute of Standards and Technology. 11

OC-SVM One-class Support Vector Machines. 46, 47, 52, 58, 61, 72, 77, 78

OS Operating System. 2, 4, 35, 40, 50, 64, 65, 67, 68, 70, 71, 72, 74, 80

PC Program Counter. 29

PCA Principal Component Analysis. xii, xiii, xv, 35, 51, 52, 54, 55, 57, 59, 72, 75, 76

PCI Peripheral Component Interconnect. 67

PMU Performance Monitoring Unit. iii, 2, 24, 25, 26, 27, 28, 33, 34, 41, 81

PULP Parallel Ultra Low Power. 52

RaaS Ransomware as a Service. 9

RADICS Rapid Attack Detection, Isolation and Characterization Systems. 36

RISC Reduced Instruction Set Computer. 3, 27

RISC-V Reduced Instruction Set Computer V. iii, 3, 27, 34, 45, 48, 52, 65, 67, 68, 70, 71, 72, 73, 75, 81

ROC Receiver Operating Characteristic. 19, 37

ROP Return-Oriented Programming. 13, 24

RSA Rivest–Shamir–Adleman. xv, 52, 53, 54, 55, 56, 57, 59, 60, 61

SBO Stack Buffer Overflow. xiv, 4, 13, 45, 46, 49, 51, 52, 53, 80

SGD Stochastic Gradient Descent. 37

SHA Secure Hash Algorithm. 52, 53, 55, 56, 58, 61

SLICC Specification Language including Cache Coherence. 66

SPARC Scalable Processor Architecture. 67

SVM Support Vector Machines. 36, 37, 47

TDT Threat Detection Technology. 36

TLB Translation Lookaside Buffer. 28, 32, 33

TN True Negative. 19

TP True Positive. 19

TPR True Positive Rate. 19, 39

UART Universal Asynchronous Receiver-Transmitter. 67

VirtIO Virtual Input-Output. 71

VirtIOMMIO Virtual Input-Output & Memory-Mapped Input-Output. 71

Contents

List of Tables xii

List of Figures xiv

1 Introduction 1

2 Malware fundamentals 5
- 2.1 Brief historical review . . . 5
- 2.2 Malware properties and classification . . . 9
  - 2.2.1 Malware goals . . . 11
  - 2.2.2 Vulnerability exploitation . . . 13
  - 2.2.3 Propagation method . . . 14
  - 2.2.4 Concealment strategy . . . 15
  - 2.2.5 Zero-day malware . . . 16
- 2.3 Overview of malware detection . . . 18
  - 2.3.1 Evaluation metrics . . . 18
  - 2.3.2 Software-based malware detection . . . 20
  - 2.3.3 Hardware-based malware detection . . . 22

3 Hardware-based malware detection approaches 26
- 3.1 Performance Monitoring Units . . . 27
  - 3.1.1 CPU microarchitecture concepts . . . 29
  - 3.1.2 Intel PMU . . . 33
  - 3.1.3 AMD PMU . . . 33
  - 3.1.4 ARM PMU . . . 33
  - 3.1.5 RISC-V PMU . . . 34
- 3.2 Hardware-based detection framework . . . 34
- 3.3 Hardware-based detection assessment . . . 36
  - 3.3.1 Performance . . . 37
  - 3.3.2 Efficiency . . . 39
- 3.4 Machine learning techniques considerations . . . 41

4 Zero-day hardware-based detection of stack buffer overflow attacks 45
4.1 Anomaly detection enabling zero-day malware detection 46
4.2 Methodology 48
4.2.1 Simulation environment 49
4.2.2 Hardware-based malware detection framework 51
4.3 Results 52
4.3.1 Feature selection 54
4.3.2 Performance in stealthy mode 54
4.3.3 Performance in overt mode 55
4.4 Brief exploration of autoencoders 60

5 Zero-day hardware-based detection of malware in operating systems 64
5.1 gem5 simulator 65
5.2 Methodology 68
5.2.1 Simulation environment 70
5.2.2 Hardware-based malware detection framework 71
5.3 Results 72
5.3.1 Feature selection 75
5.3.2 Performance 76

6 Conclusions 80

7 List of publications 82

8 Code availability 83

Bibliography 84

List of Tables

2.1 Malware goals and its properties. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 11
2.2 Common exploitable malware vulnerabilities [60, 61]. 14
2.3 Confusion matrix. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 19
2.4 Most common metrics for performance evaluation of classification and malware detection. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 20
3.1 Dominant CPU manufacturer/design companies, their Instruction Set Architectures (ISAs) and end-user markets. 27
3.2 Reference studies in HMD, including the full list of targets, classifiers tested, and reference systems. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 38
3.3 Summary of best-case performance from reference studies in HMD. HPCs column refers to the number of hardware events the classifiers consider. A is Accuracy, P is Precision, S is Specificity, and F1 is the F1-Score. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 39
3.4 Performance and efficiency of ML classifiers when applied to HMD [94]. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 40
4.1 Reference studies in anomaly HMD. HPCs column refers to the number of hardware events the classifiers consider. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. 48
4.2 Result of feature selection, with HPCs remained after manual pruning and ranked according to Principal Component Analysis (PCA). Index 1 defines the most significant counter. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. 54
5.1 Benign applications and malware employed as benchmarks. 74
5.2 Number of simulations executed. 75

5.3 Result of feature selection, with HPCs remained after manual pruning and ranked according to PCA. Index 1 defines the most significant counter. 76

List of Figures

1.1 Overview of hardware-based malware detection approaches. 3
2.1 Main milestones and malware in the history of malware. 7
2.2 Zero-day malware lifecycle [7]. 17
2.3 Overview of contemporary solutions for malware detection. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 18
2.4 Example of patterns in architectural and micro-architectural hardware events reflecting the program phases, enabling distinguishing between benign and malicious applications [88]. 23
3.1 Performance Monitoring Units. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 28
3.2 Microprocessor without Interlocked Pipelined Stages (MIPS) pipeline stages [125]. 30
3.3 Pipelined CPU with multiple functional units [129]. 31
3.4 A virtual memory system [125]. 32
3.5 Generic hardware-based detection framework. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY. 35
3.6 Machine learning techniques employed to improve the performance of HMD approaches. 44
4.1 Canonical Stack Buffer Overflow (SBO) attack flow. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . 46
4.2 Anomaly novelty detection [175]. 47
4.3 Methodology overview for the zero-day detection of SBO attacks. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. 49
4.4 Execution modes configuring different attack scenarios. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . 50
4.5 Setup for simulations of zero-day detection of SBO attacks. 53
4.6 Accuracy for Stealthy mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. 56

4.7 Accuracy for Rivest–Shamir–Adleman (RSA) with constant prime number, in Stealthy mode. IF is an Isolation Forest, and EE is an Elliptic Envelope. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . . 57
4.8 RSA related PCA decomposition in the three first principal components, in Stealthy mode. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . . 57
4.9 Accuracy for Overt mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . . 58
4.10 Accuracy for RSA with constant prime number, in Overt mode. IF is an Isolation Forest, and EE is an Elliptic Envelope. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . 59
4.11 RSA related PCA decomposition in the three first principal components, in Overt mode. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY. . . . 59
4.12 Anomaly HMD framework including the autoencoder. . . . 61
4.13 Autoencoder architecture for eight features. . . . 62
4.14 Accuracy when the autoencoder is inserted in the framework, for Stealthy mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers. . . . 63
5.1 Amount of new malware by year for Windows, Linux, and Android [186]. . . . 65
5.2 Overview of `gem5`'s architecture [188]. . . . 66
5.3 `gem5` execution modes [188]. (a) System-call emulation mode. (b) Full-system mode. . . . 67
5.4 `gem5` hardware with RV64GC for full-system execution mode, based on HiFive platform [189]. . . . 68
5.5 `gem5` software layers with RV64GC for full-system execution mode [189]. . . . 69
5.6 Methodology overview for the zero-day detection of malware in operating systems. . . . 70
5.7 Benchmark execution modes. . . . 72
5.8 Setup for simulations of zero-day detection of malware in the operating system. . . . 73
5.9 Accuracy in detecting malware in applications running in the operating system (clustered by application). IF is an Isolation Forest, and EE is an Elliptic Envelope. . . . 77
5.10 Accuracy in detecting malware in applications running in the operating system (clustered by classifier). . . . 79

[empty]

Chapter 1

Introduction

Contemporary society is highly dependent on computers. They are used in a multitude of applications in many areas, such as e-learning and online courses, automation of business, telemedicine, research and development in science and engineering, online banking, gaming for entertainment, email and video conferencing, traffic management, simulations in military and defense, and home automation. Some applications are also crucial and sensitive, like life-supporting, control of nuclear power plants, distribution of electrical power to cities, voting and election systems, infrastructure for telecommunications, and aerospace transportation.

Security problems have arisen alongside computer evolution. The attackers (also called hackers), central characters together with users and security experts, have diverse motivations to perpetrate computer crimes: revenge, disgruntled employees may take their frustration out on the computer; political or ideological conviction, like activists seeking to disrupt operations of groups they disagree; governmental and corporate espionage, like some company stealing information from competitors to gain an advantage; and finally, the main reason for most of the cybercrimes, the financial gain. Accessing bank accounts or credit card information gives a short path to the money. However, accessing intellectual properties and personal and company information also has a commercial value that can generate financial and operating revenue. According to Cybersecurity Ventures, the world's leading researcher for the global cyber economy, global cybercrime will be more profitable than the global trade of all major illegal drugs combined [1].

Security has become paramount and one of the most important areas in computing. The World Economic Forum's Global Risk Report 2023 ranks cyber insecurity as the eighth most significant global risk, alongside major threats such as climate change and involuntary migration [2]. Cybersecurity Ventures estimated the annual global cybercrime cost of USD $9.5 trillion in 2024 and USD $10.5 trillion in 2025 [3]. On the defense side, they predicted global spending on cybersecurity products and services of more than USD $1.75 trillion cumulatively from 2021 to 2025, growing 15 percent year-over-year [4].

Understanding the security threats is an important step forward. We may classify them into two large classes: physical security threats and malware. The first comprises side-channel attacks [5] and those threats related to the computer facilities, as simple as the possibility of theft, fire, floods, and electromagnetic interference. Side-channel attacks are methods to deduce sensitive information by exploiting timing, power consumption, and electromagnetic and acoustic emission characteristics. Malware, short for malicious software, is the threat this thesis targets. It includes any code modification within a software system aimed at causing harm or disrupting the system's intended function. Malware attacks cover spying, intrusive ads, email abuse, system damage, ransom demands, data release, slowdown, browser manipulation, and unauthorized access to sensitive information. The damage caused by malware goes beyond the economic losses. In 2017, a major hospital in Dusseldorf, Germany, reported the first death attributed to a ransomware attack, the prevalent malware threat. The resulting failure in the information technology system forced the transfer of a woman, who required urgent hospitalization, to another city, ultimately leading to her death [6].

The complexity and size of modern computers, often indicated by a rising number of lines of code, amplify the malware threat. Factors such as numerous bugs, unsafe programming languages, improper configuration, and the ease of concealing malicious code create potential vulnerabilities. Additionally, the increased network connectivity expands the security risks, making all computer devices potential targets for attackers. The first step towards a secure cyber environment is malware detection, which currently has two main branches: software-based and hardware-based approaches. The software-based approach is the most widespread, relying on software tools and techniques to identify and analyze the presence of malware. The hardware-based approach, also called Hardware-Based Malware Detection (HMD), is the latest attempt to boost malicious code detection. HMD involves dynamically analyzing architecture and microarchitecture events in a Central Processing Unit (CPU) using Machine Learning (ML) algorithms to distinguish between benign applications and malware (see Figure 1.1). This thesis investigated this approach in depth, focusing on how the core component - the ML methods - should be employed to optimize performance and efficiency.

This thesis gives a couple of contributions to the field of cybersecurity, specifically for malware detection using ML: (i) it deepens and systematizes knowledge about malware, its properties, classification, and related concepts; (ii) it discusses and compares the main available approaches for malware detection, defining their weaknesses and strengths; (iii) it deepens and systematizes knowledge about the HMD approaches, identifying their key features and addressing the CPUs' Performance Monitoring Units (PMUs), starting point that enables the detection by hardware; (iv) it reviews several distinct papers in HMD, trying to elaborate a landscape of its targets in terms of malware threats, ML methods, devices and Operating Systems (OSs). The proposed solutions are evaluated in terms of performance and

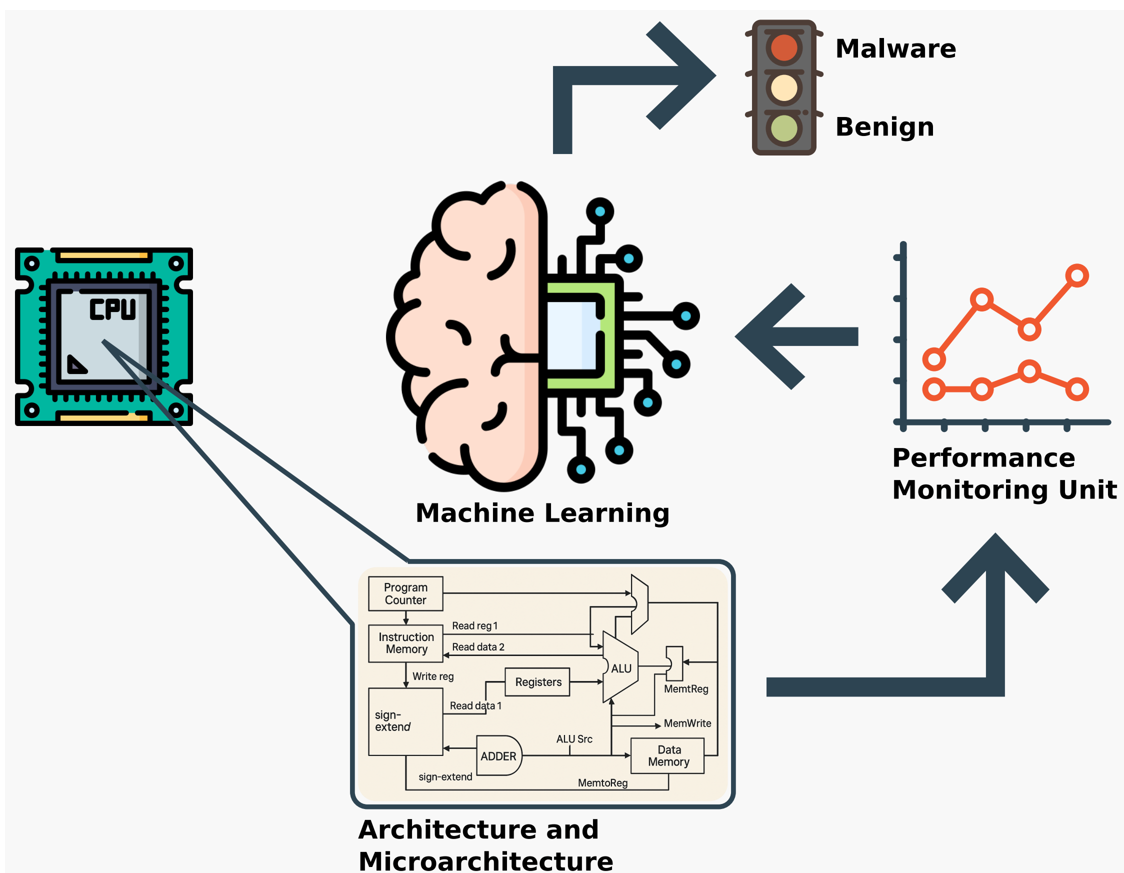


Figure 1.1: Overview of hardware-based malware detection approaches.

efficiency whenever the data is available to understand the HMD potentials and limitations. (v) The two case studies detailed in this thesis investigated the HMD performance capabilities in detecting the challenging zero-day malware [7, 8, 9, 10], a threat that exploits previously unknown vulnerabilities. Moreover, these studies implement HMD schemes in Reduced Instruction Set Computer V (RISC-V) [11] in an attempt to disseminate the hardware-based malware detection approaches to the community surrounding this platform. RISC-V is an emerging platform attracting attention due to its simplicity, modularity, extensibility, and royalty-free open-source licenses. It is a standard Instruction Set Architecture (ISA) based on established Reduced Instruction Set Computer (RISC) principles.

However, the two case studies detailed present a delimited scope, do not cover all aspects of HMD and ML methods, and leave room for further studies. (i) The hardware-based detection capability is evaluated solely by the performance. (ii) Deep learning, an important branch of ML, is not analyzed in the experiments. (iii) To address zero-day malware, the detectors employed anomaly detection [12], which involves identifying patterns that deviate from expected behavior.

The widely used ML methods for profiling typical behavior have not been tested. This thesis is organized as follows: Chapter 2 discusses the malware fundamentals, including an overview of detection. Chapter 3 delves into the relevant aspects of HMD approaches. Chapter 4 presents a case study of the detection of Stack Buffer Overflow (SBO) attacks using the hardware-based approaches. Chapter 5 presents a case study of detecting malware running in Linux, expanding the discussion to the environment most conducive to the malware spreading, the OSs. Chapter 6 closes with the concluding remarks. Finally, Chapter 7 shows the papers published during this PhD study.

Chapter 2

Malware fundamentals

Adapted, with permission, from Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo, "A Survey on Hardware-Based Malware Detection Approaches," in IEEE Access, 2024, doi: 10.1109/ACCESS.2024.3388716. Licensed under CC-BY.

Every year, a massive quantity of malware risks businesses and individuals [13]. Successful attacks lead to consequences that can be categorized into four groups: (i) unauthorized disclosure, where an authorized entity gains access to data; (ii) deception, where an authorized entity receives false data; (iii) disruption, causing interruptions in system services; and (iv) usurpation, resulting in unauthorized control of system services [14].

The first step toward secure systems is understanding the relevant aspects of the broad class of security threats called malware and its detection possibilities. Section 2.1 creates a timeline, highlighting the main milestones and malware in history. Section 2.2 dissects the malware properties and classification, providing a strong foundation to understand the field. Finally, Section 2.3 reviews relevant aspects and approaches to the challenge of malware detection.

2.1 Brief historical review

There is a vast literature on the history of malware, [15, 16, 17] are just a few examples. Its review allows to build a timeline of main milestones and malware in the field, as shown in Figure 2.1. We may start back to 1971 when Bob Thomas and Ray Tomlinson created an experimental program called **Creeper**, designed to self-replicate between computers inside the Arpanet. Although Creeper is considered the first worm, it is not malware since it did not cause damage or disruption. In 1982, at just 15 years old, Rich Skrenta wrote a program to play pranks on his friends, dubbed **Elk Cloner**. Besides being the first virus for Apple computers,

the program was the first known malware to escape its creation environment. Elk Cloner quickly spread among Skranta's friends, transferring itself from the memory computer to a floppy disk and then to another computer. Deliberately malicious, the virus could inadvertently write over and erase the floppy disks.

While the effects of Elk Cloner were contained on Skranta's computer friends, the first globally spread malware was the **Brain virus** in 1986. It was also the first to IBM/Microsoft computers and the first to conceal its existence on the disk to evade detection. Created in Pakistan by the brothers Amjad and Basit Farooq Alvi from a medical software distributor, it was initially developed to prevent copyright infringement. When installed, Brain would display a message prompting pirates to call the brothers to receive the vaccination. In 1987, there was an important milestone. Fred Cohen published the first paper about malware in the scientific community [18], discussing viruses' primary attack mechanisms and protection possibilities.

The world's first worm was born in 1988. Called **Morris**, in allusion to its author, Robert Morris, it was created not for malicious intent but as a proof of concept about malware self-propagation and self-replication. The worm proved to be much more effective than anticipated. It quickly spread, copying itself repeatedly and infecting 10% of all computers connected to the internet. Because the worm not only copied itself to other computers but also copied itself repeatedly on infected computers, it unintentionally ate up memory and brought multiple computers to a grinding halt. The incident caused damages and financial losses. Jumping to 1999, the **Melissa worm** showed how fast malware can spread by email. The fastest-spreading worm for its time caused major overloads on Microsoft Outlook and Microsoft Exchange email servers, resulting in slowdowns at several corporations. Estimations are that one million email accounts and at least 100,000 computers were infected, costing 80 million dollars for damage clean-up and repair.

In 2000, the **I Love You worm** innovated using ingredients of social engineering and phishing to propagate by email. At 24 years old, the Philippines resident Onel de Guzman found himself unable to afford dial-up internet service and built the worm to steal other people's passwords. De Guzman used psychology to prey on people's curiosity and manipulate them into downloading malicious email attachments disguised as love letters. Infecting millions of Windows computers worldwide within a few hours of its release, I Love You caused damage by deleting files and briefly shutting down the United Kingdom's Parliament's computer system. Other famous worms marked the period from the next four years: **Code-Red**, in 2001, targeted Microsoft web servers, launching Denial of Service (DoS) attacks and displaying a message of hacking; **Slammer**, in 2003, attacked Microsoft SQL servers, causing DoS attacks on some Internet hosts and dramatically slowing general internet traffic; **Witty**, in 2004, exploited the IBM Internet Security Systems, generating network traffic and writing in the hard disk until the machine is rebooted or permanently crashes; **Sasser**, in 2004, targeted Microsoft Windows XP and 2000,

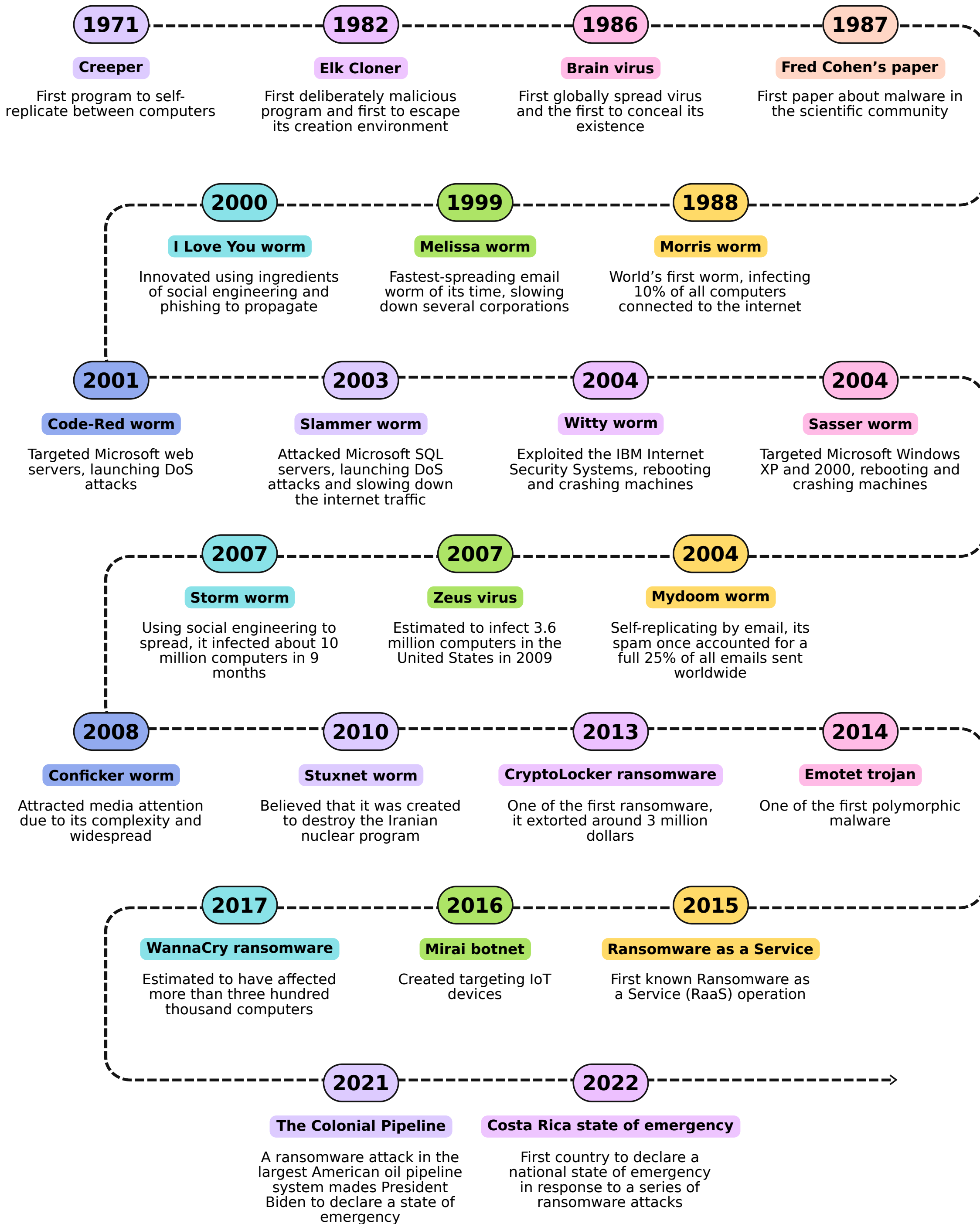


Figure 2.1: Main milestones and malware in the history of malware.

causing computers to crash and reboot repeatedly.

Yet in 2004, the **Mydoom worm** also used email to self-replicate and infect Microsoft Windows around the world. Astonishingly effective, its spam once accounted for a full 25% of all emails sent worldwide and ended up causing 35 billion dollars in damages. Mydoom also used infected computers to create a botnet and launch distributed DoS attacks. In 2007, the **Zeus virus** infected Microsoft Windows computers through phishing and drive-by-downloads schemes. It demonstrated the dangerous potential of a trojan-style virus that can deliver many different types of malicious software. The virus was often used to steal banking information by keystroke logging and form grabbing. Zeus was estimated to infect 3.6 million computers in the United States in 2009. Another significant worm from 2007 was **Storm**, which infected about 10 million computers in 9 months. It was a fast-spreading email spamming threat to Microsoft operating systems. Like I Love You, it used social engineering to spread, except in this case with fear and horror instead of love, through messages like "230 dead as storm batters Europe". Its variants install a rootkit component or merge into a botnet.

In 2008, the **Conficker worm** arose, one of the most notorious malware ever created due to its complexity and widespread. It infected millions of Microsoft computers in over 190 countries, including government, business, and home computers. It encrypted its payload, modified its code frequently, used a complex algorithm to generate random domain names, spread through the network and infected USB drives, delayed activation, and behaved as a botnet to enable a range of malicious purposes. The media attention garnered by Conficker helps further raise the idea of network security in the public consciousness. When the **Stuxnet worm** was found in 2010, it had been spreading undetected for about a year and had already done what it was built for. It is believed that Stuxnet was created to destroy or at least slow down the Iranian nuclear program. It physically sabotaged turbines for uranium enrichment by changing rotation frequencies.

In 2013, one of the first ransomware variants was born: **CryptoLocker**. Distributed via infected email attachments and botnets captured by the Zeus virus, it systematically encrypts data on infected computers. To regain access to these encrypted resources, the requested price for decryption was two bitcoins. In 2014, an operation managed to seize control of the malicious botnet and decrypt the hostage data free of charge. It is believed that the operators of CryptoLocker successfully extorted around 3 million dollars from their victims. The prime example of what is known as polymorphic malware arose in 2014, the **Emotet trojan**. As discussed in Sub-section 2.2.4, polymorphic malware works by slightly altering its code every time it reproduces, creating not an exact copy, making its detection difficult. Emotet persists as a modular program used to deliver other forms of malware and is often shared through traditional phishing attacks.

With the rise of the Internet of Things (IoT), smart IoT devices present a vast new wave of vulnerabilities. In 2016, the **Mirai botnet** was created, primarily targeting online consumer devices such as cameras and home routers running Linux.

It was used in some of the largest and most disruptive distributed DoS attacks. In 2017, the famous **WannaCry ransomware** was devised. It spread worldwide using an exploit developed by the United States National Security Agency (NSA) for Microsoft Windows. It is estimated to have affected more than three hundred thousand computers in about 150 countries and infected many hospitals, banks, telecommunication companies, warehouses, and industries.

In recent years, ransomware malware has tapered off. However, hackers are targeting more high-profile targets and causing more significant damage. **Ransomware as a Service (RaaS)** is a troubling trend that has gained momentum in recent years. Offered on dark web marketplaces, it provides a plug-and-play protocol in which professional hackers conduct ransomware attacks in exchange for a fee. RaaS empower anyone without technical skills but with ill intent and money to spend [19, 20].

One example of an attack on high-profile targets happened in 2021. **The Colonial Pipeline**, the largest American oil pipeline system responsible for 45% of the eastern United States' gasoline and jet fuel, suffered a ransomware attack that afflicted computerized equipment managing the pipeline. The attack lasted for several days and prompted President Biden to declare a temporary state of emergency [21]. In 2022, **Costa Rica** suffered a series of ransomware attacks. First, the Ministry of Finance was crippled, and a following attack took the nation's healthcare system offline. Costa Rica made history as the first country to declare a national state of emergency in response to a cyberattack [22].

2.2 Malware properties and classification

While early malware exhibited simple behavior, with its evolution over time, it became blended, with growing complexity and diverse properties. Blended malware uses multiple methods and techniques to maximize its ability to spread, hide, and perform various actions. Some malware even supports an update mechanism that allows it to change its behavior once deployed. This scenario imposes a difficulty in malware classification, requesting a cyberattack taxonomy. Although there is no consensus in the scientific community, some attempts to develop this taxonomy exist. In most cases, they also comprise physical threats.

Bishop [23] presented a taxonomy of Unix vulnerabilities, classifying them into six axes: nature of the flaw, time of introduction (when), exploitation domain (what is gained), effect domain (what is affected), minimum number of components necessary and source of identification of the vulnerability. **Howard and Longstaff** [24] proposed a broad and process-based taxonomy, considering factors such as attacker motivation and objectives. It consists of five stages: attackers (e.g., hackers, terrorists), tools used, access gained, results (e.g., corruption, disclosure of information), and objectives (e.g., inflicting damage, gaining status). However, Howard

and Longstaff's taxonomy lacks practical value since it considers the whole attack process instead of concerning the attack itself.

Lough and Davis [25] proposed a taxonomy based upon four characteristics of attacks: improper validation (insufficient or incorrect validation results in unauthorized access to information or a system); improper exposure (a system or information is improperly exposed to attack); improper randomness (insufficient randomness results in exposure to attack); and improper deallocation (information is not properly deleted after use and thus can be vulnerable to attack). While this taxonomy can classify any attack using these four characteristics, it is general and does not help to deal with attacks in daily tasks.

Hansman and Hunt [26] attempted to suggest a more pragmatic taxonomy than the previous ones. The idea is to classify the attacks in four dimensions. The first is the attack vector, related to the method by which an attack reaches its target. The attack target is covered in the second dimension. The third dimension is related to the vulnerabilities and exploits. They proposed to use in this dimension the list defined by the Common Vulnerabilities Exposures (CVE) program [27], and that, due to its importance, is introduced here as a brief parenthesis. CVE lists publicly disclosed cybersecurity vulnerabilities identified in real-world software or hardware. It is managed by the Mitre Corporation, a non-profit organization that supports United States government agencies from various fields, including cybersecurity. Technical professionals use CVE records to ensure they are discussing the same issue and to coordinate their efforts to prioritize and address the vulnerabilities. Finally, the fourth dimension of Hansman and Hunt is the attack payload, which refers to the attack actions and effects beyond itself. This taxonomy also allows for the possibility of further dimensions, which, although not necessary, may enhance the knowledge of the attack. The authors describe in detail how to use each dimension, making their proposal one of the most useful.

Kjaerland [28] focused on how commercial and government sectors experience different types of attacks. They suggested four facets to organize attacks: method of operation, which refers to the methods used by the perpetrator to carry out an attack; impact, which refers to the effect of the attack; source, which refers to the source of the attack; and target, which refers to the victim. Kjaerland taxonomy also has limited applicability in day-to-day routines since it has a high-level view of the taxonomy of operations, but flaws in recognizing the foundations of the attack.

More recently, **Harry and Gallagher** [29] proposed a taxonomy where any given attack can have one of two primary objectives: the disruption to the functions of the target organization or the illicit acquisition of information. The disruptive objective is divided into five categories: message manipulation, external DoS, internal DoS, data attack, and physical attack. In turn, the goal of illicit acquisition of information is also divided into five categories: exploitation of sensors, exploitation of end hosts, exploitation of infrastructure, exploitation of application servers, and exploitation of data in transit.

Other taxonomies target specific domains. **Mirkovic and Reiher** [30] elaborate a taxonomy for distributed DoS attacks and defense mechanisms. **Zhu and Sastry** [31] for attacks in industrial control systems, and **Gruschka and Jensen** [32] for attacks in cloud services. Facing the absence of a single standard taxonomy, this thesis dissects software threats in terms of goals and three useful properties for a practical approach. Table 2.1 relates **malware goals** and the three properties: **vulnerability exploitation**, **propagation method**, and **concealment strategy**. Moreover, a broad concept related to vulnerability exploitation is the **zero-day malware**, which is discussed at the end of this section.

Table 2.1: Malware goals and its properties. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

Malware goal	**Vulnerability exploitation**	**Propagation method**	**Concealment strategy**
Virus	XSS, memory corruption, overflow, file inclusion	File-based transmission	Polymorphism
Worm	XSS, memory corruption, overflow, file inclusion	Network-based transmission	Polymorphism or metamorphism
Trojan horse	XSS, memory corruption, overflow, path traversal, file inclusion	Social engineering	No concealment
Spyware	XSS, memory corruption, overflow, file inclusion, XXE	Internet download	Encryption
Adware	XSS, open redirect, SSRF	Software bundling	No concealment
Ransomware	SQL injection, path traversal, file inclusion, XXE, SSRF	Email attachment	Encryption
Backdoor	SQL injection, path traversal, file inclusion, XXE, SSRF	Remote access	Encryption
Keylogger	XSS, file inclusion, SSRF	File-based transmission, social engineering (phishing)	Encryption
Botnet	SQL injection, memory corruption, overflow, file inclusion, SSRF	Network-based transmission, social engineering	Encryption and polymorphism
Rootkit	SQL injection, memory corruption, overflow, file inclusion, SSRF	Kernel-level exploit, virtual machine	Obfuscation

2.2.1 Malware goals

Malware is the umbrella term describing threats with different goals. Organizations like National Institute of Standards and Technology (NIST) [34] and European Union Agency for Cybersecurity (ENISA) [35] recognize the following keywords commonly used to describe the malware goals:

- **Virus** is the malicious code capable of inserting itself into other programs. The term alludes to biological processes in which an infectious agent attaches itself to some living organism. In a computer, once the virus is executing, it can perform any function, such as file corruption, deletion, and data theft [14, 15];

- **Worm** is the malicious code that propagates similarly to viruses but does not require a target software to replicate, often exploiting software vulnerabilities in network connections, shared media, email, and instant messenger. A feature that distinguishes them from viruses and trojan horses is their propagation speed. They can infect in a matter of hours or even minutes in millions of computers [14, 15, 36];

- **Trojan horse** is the malicious code masquerading as a useful program. It is an allusion to Greek mythology when a giant wooden horse was used by the Greeks to invade Troy. An example of an insidious type of Trojan horse is an anti-virus program whose execution intentionally introduces viruses onto the computer [14, 15, 37];

- **Spyware** is the malicious code secretly installed into an information system to transmit private user data to an external entity. The difficulty of getting spyware rid of makes the problem worse [38, 39, 40];

- **Adware** is the malicious code that displays computer advertisements, primarily aiming for financial benefits. Some adware is voluntarily installed by computer owners on their machines as free software. They are one of the less harmless attacks, sometimes innocuous or merely annoying [38, 41, 42];

- **Ransomware** is the malicious code that denies access to a user's data, usually encrypting it until a ransom is paid. In recent years, ransomware has been one of the most notorious malware. Due to the expressive financial losses, it has become a very profitable business for criminals and a serious threat to organizations. This economic impact is also reflected in the high interest of the scientific community in this threat [43, 44, 45, 46];

- **Backdoor** is the malicious code that opens systems to external entities by subverting local security policies to allow remote access and control over a network. Programmers usually use backdoors legitimately to debug and test programs. For example, programmers may wish to avoid all the necessary setup and authentication to debug an application under development with an authentication procedure or a long setup. Then, a backdoor is used to recognize some special sequence of input, a specific user identity, or an unlikely sequence of events. Backdoors become threats when unscrupulous programmers use them to gain unauthorized access. Currently, a backdoor is usually implemented as a network service listening on some non-standard port that the attacker can connect to, and issue commands through to be run on the compromised computer [14, 47];

- **Keylogger** is the malicious code designed to record keystrokes, used to obtain passwords or encryption keys to bypass security measures [48, 49, 50];

- **Botnet** is a network of infected computers controlled by a remote criminal. Botnets are typically planted on hundreds or thousands of computers belonging to unsuspecting third parties to perform malicious tasks like distributed DoS, sending spam, traffic sniffing, keylogging, and spreading new malware [51, 52, 53];
- **Rootkit** is the malicious application attackers use to conceal their activities and maintain control over a host. The strategies employed through time are associated with hiding rootkits, one layer below it can be detected. The early rootkits worked in user mode, modifying utility programs and libraries. The next generation moved down a layer, making changes inside the kernel and co-existing with the operating system code. The latest generation uses a virtual machine monitor or hypervisor to run the operating system inside, allowing the rootkit to run entirely below the visibility of even kernel code [54, 55, 56].

2.2.2 Vulnerability exploitation

A precise definition states software vulnerability as a weakness that could be exploited or triggered by a threat source [34]. The scientific community has been discussing it for several years [26, 57, 58, 59], but the takeaway lacks a practical system to deal with the vulnerabilities. The answer for this lack may be provided based on the CVE and Common Weakness Enumeration (CWE) lists. The CWE is a list of common software and hardware weaknesses, conditions that can lead to the introduction of vulnerabilities. Also maintained by the Mitre Corporation, CWEs are more abstract than CVE list and describe classes of weaknesses. Table 2.2 presents a compilation of common exploitable malware vulnerabilities, elaborated based on two specific references: the CVE Details [60], a comprehensive cybersecurity vulnerability database from company SecurityScorecard; and the CWE community report regarding the 2024 top dangerous software weaknesses [61]. The CWE identifier is used just as a reference, and some vulnerabilities may be present also in others CWEs.

Although these represent most vulnerabilities exploited by malware, the CWE list details many others. Related to the widely diffused memory corruption, it originates two types of attacks that frequently appear in HMD studies: (i) **control-flow attacks** and (ii) **data-only attacks**. The first [62] are common, easy to construct, and demand minimal application-specific knowledge. They deviate the program's execution flow to specific memory locations containing malicious code. SBOs [63], code injection [64], Return-Oriented Programming (ROP) [65] and Jump-Oriented Programming (JOP) [66] are some techniques used to implement control-flow attacks. Conversely, data-only attacks [67] are rarer, subtler, and require advanced knowledge of program semantics. They do not deviate the code nor inject additional code. They manipulate or corrupt essential data elements, such as identification or

Table 2.2: Common exploitable malware vulnerabilities [60, 61].

Vulnerability	Identifier	Description
Cross-Site Scripting (XSS)	CWE-79	It occurs when a web application does not properly neutralize or validate user-controllable input. It allows attackers to inject malicious scripts into the web application viewed by other users
Memory corruption	CWE-119	It happens when a program can read from or write to a memory location outside the buffer's intended boundary. It may be linked to other variables, data structures, or internal program data to execute an attack
Overflow	CWE-120	It occurs when the program can copy the input buffer to the output buffer without checking that the input buffer size is less than the output buffer size. If the buffer overflow happens, it may compromise the security
SQL injection	CWE-89	It is a web vulnerability that enables an attacker to manipulate the queries that an application makes to its database. It happens when the user input is improperly neutralized or validated, allowing attackers to inject malicious SQL code into the query
Cross-Site Request Forgery (CSRF)	CWE-352	It is another web vulnerability in which the application does not sufficiently verify whether a request was intentionally provided by the user who submitted the request. By exploiting the authenticated session, an attacker can compromise the security
Input validation	CWE-20	It occurs when the program fails to properly validate inputs or data, leading to a variety of exploits
Path traversal	CWE-22	Also known as directory traversal, is a web vulnerability that exploits improper validation of file paths. Many file operations are intended to be confined within a restricted directory, but this vulnerability allows attackers to manipulate file paths to access files or directories outside the intended restricted directory
File inclusion	CWE-98	It is a web vulnerability where the application allows user input to specify a file to include or execute but fails to validate or sanitize the input properly. It can result in an attack
Open redirect	CWE-601	It is another web vulnerability in which the application accepts a user input that specifies a link to an external site and uses that link in a redirect. A security compromise may happen
XML external entity (XXE)	CWE-611	It occurs when applications that parse XML do not validate or sanitize the input and allow the injection of unsafe XML entities, potentially leading to security attacks
Server-Side Request Forgery (SSRF)	CWE-918	It occurs when a web server receives requests but does not ensure which is being sent to the expected destination, enabling access to internal resources, sensitive data, or interaction with external systems

configuration data, influencing target application behaviors during runtime.

2.2.3 Propagation method

The propagation method refers to how malware spreads to reach the desired targets. It is also mentioned as an infection vector. File-based transmission, network-based transmission, and social engineering are broad and general propagation methods [14, 68]. They are quickly discussed in the following, together with more specific methods.

- **File-based transmission**: the malicious code is added in a file that is shared between two or more computers;
- **Network-based transmission**: the malicious code uses the network to

reach other computers, including wireless networks. It is related to the exploitation of software vulnerabilities in the network layer;

- **Social engineering**: human psychology is exploited to trick users into enabling malicious code execution. For example, when the user permits the installation and execution of some trojan horse, or in phishing when social engineering is employed to get access to sensitive information (usernames, passwords, financial details, and others);
- **Internet download**: the malicious code is downloaded and installed on the computer when the user visits a malicious web page without their knowledge or consent;
- **Software bundling**: the malicious code is packed together and installed simultaneously with legitimate software;
- **Email attachment**: the malicious code is embedded in an email attachment;
- **Remote access**: the malicious code uses the possibility of remote access to infiltrate computers. It is related to the network-based transmission, which enables the remote access;
- **Kernel-level exploit**: the malicious code uses the kernel-level to conceal it activity, which is one level below it can be detected;
- **Virtual machine**: virtualization technique is becoming a standard technique for business. One computer or server runs multiple operating systems or multiple sessions of an operating system at the same time. The malicious code may use the host of the virtual machines to perform the attack, concealing its activity from the virtual machine's security.

2.2.4 Concealment strategy

The concealment strategies (also known as camouflage) [69, 70, 71] refer to techniques that malware may employ to hide their tactics from users and detection mechanisms. They are critical in prolonging the lifetime of the malware, significantly contributing to its success. Regarding this property, malware can be classified into two main groups: (i) **no concealment** and (ii) **stealthy malware**. No concealed malicious code lacks techniques to hide itself, making it easy to detect. As shown in Table 2.1, only a tiny subset of threats does not employ concealment. Traditional viruses may not heavily focus on concealment, spreading by attaching themselves to executable files. Similarly, simple trojan horses may prioritize their primary goal over camouflage. Adware, due to its advertising nature, is commonly in plain sight.

Conversely, stealthy malware [69, 72, 73] is a general term for all kinds of malicious code that actively takes steps to evade detection mechanisms. They spread dormant in the wild for extended periods, gathering sensitive information before a suitable detection succeeds. In general, stealthy malware aims to hide its trail or code. Various techniques may be employed:

- **Obfuscation** is the practice of making textual data and binary hard to understand or even unintelligible. At the first moment, it was used to protect intellectual property, thwarting reverse engineering. Later, obfuscation became the cheapest and easiest solution to hide malicious code from detection mechanisms. The most prevalent obfuscation techniques are dead code insertion, instruction replacement, register reassignment, subroutine reordering, code transposition, and code integration [70, 74, 75];

- **Encryption**, together with obfuscation, is one of the simplest techniques for concealment. An encrypted malware typically comprises a decipher and an encrypted main body with the actual malicious code. When the malware starts to run, the decipher recovers the main body. The decryptor's small size compared to the encrypted main body reduces the detection probability, making the technique feasible. Several encryption schemes may be used: basic operations (like incrementing and decrementing, bitwise rotation, and arithmetic and logical negation), using a static encryption key, using a variable encryption key, using substitution cipher, or even complex encryption algorithms [69, 70, 76];

- **Oligomorphism and polymorphism** deal with the constant decryptor employed in encryption technique, which, even small, makes possible the detection through code patterns. Oligomorphism employs a small set of decryptors, using a different one for each infection. Polymorphism, a more sophisticated attack, creates theoretically infinite variations of decryptors with the help of obfuscation methods [69, 70, 77, 78, 79];

- **Metamorphism** is the most elaborate concealment technique. The malware body is polymorphic, and there is no encryption. A new malicious code version is produced for each new infection through a mutation engine, which employs obfuscation and code transforming (register usage exchange, code permutation, code expansion, code shrinking, and garbage code insertion). Metamorphic malware is nontrivial to detect and also hard for attackers to implement correctly [69, 70, 77, 78, 79].

2.2.5 Zero-day malware

Zero-day malware [7, 8, 9, 10] is a malware that exploits previously unknown vulnerabilities. It has no history, usually has novel behavior (not seen before), and

is released before a defense mechanism is deployed. These characteristics configure a major challenge in addressing zero-day malware detection. Figure 2.2 shows the timeline of a zero-day malware [7]. It starts with a vulnerability introduced in software that is released and deployed on computers. Attackers discover the vulnerability, create a working exploit, and use it to conduct attacks on selected targets. Vendors discover the vulnerability, assess its severity, assign a priority for fixing it, and start working on a patch. The vulnerability is disclosed publicly, and a CVE identifier is attributed to it. In the case of traditional software-based malware detection (see Sub-section 2.3.2), antivirus vendors release signatures for the ongoing attack to create the detection. Antivirus vendors also release a patch for the vulnerability, and the computers that have the patch applied to them are no longer susceptible. When all computers are patched, the vulnerability ceases to have an impact.

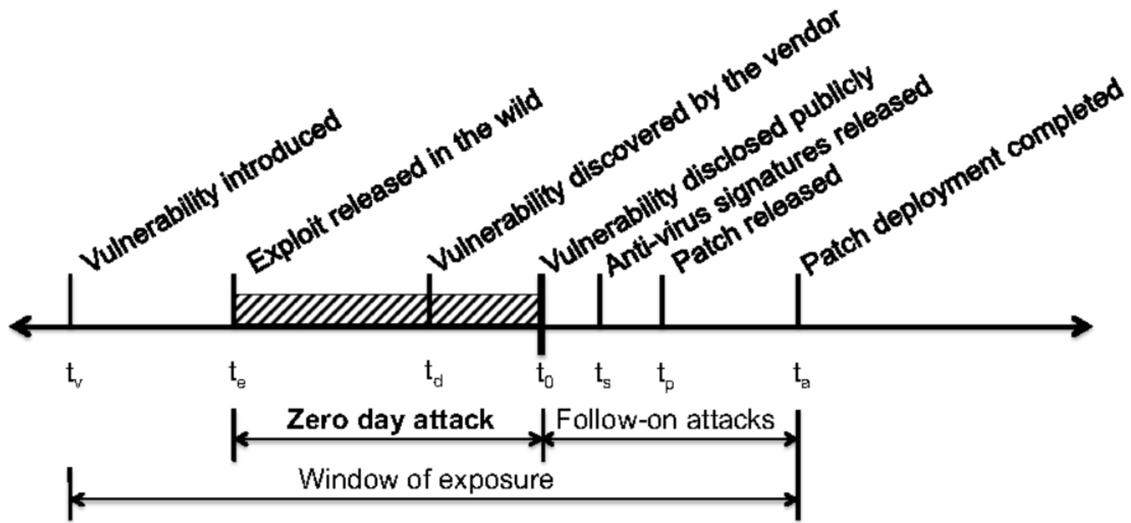


Figure 2.2: Zero-day malware lifecycle [7].

This timeline using software-based detection shows the ineffectiveness of traditional detection against zero-day malware. The delay between an attacker's exploit release and a patch deployment provides sufficient time for the malicious code action, potentially affecting many computers and causing significant damage. Zero-day malware, in conjunction with stealthy malware (see Sub-section 2.2.4), amplifies the threat. ML techniques have been developed as a solution for detecting zero-day malware [8, 9, 10]. Techniques based on ML can model more complex patterns of malware data than traditional detection.

2.3 Overview of malware detection

Malware detection consists of determining whether a given program has malicious intent. Figure 2.3 presents an overview of contemporary solutions for malware detection, splitting into two main groups: **software-based** and **hardware-based approaches**. This division arises from different system stack observation points and methodologies. The traditional detection is based on software, like the popular antivirus applications, which deal with several threats despite the name. Recent advancements increasingly rely on ML and Deep Learning (DL) techniques to boost detection [10, 9]. The hardware-based detection is more recent and arose in the 2010s.

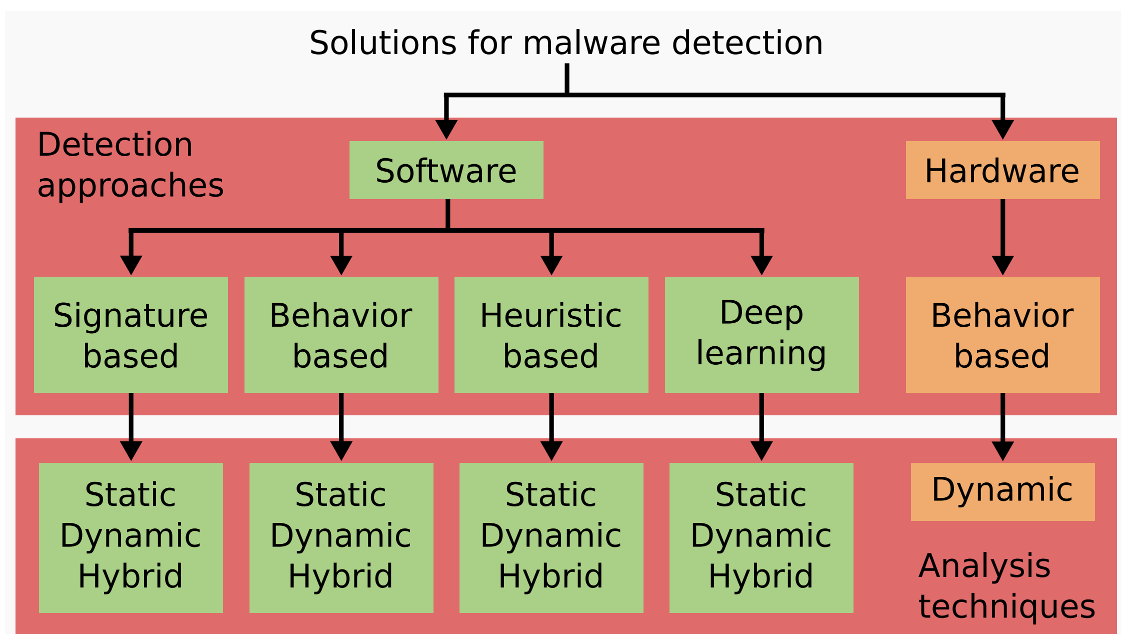


Figure 2.3: Overview of contemporary solutions for malware detection. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

The remainder of this section provides an overview of software and hardware-based malware detection (Sub-sections 2.3.2 and 2.3.3), starting with a review of metrics used to evaluate detectors (Sub-section 2.3.1).

2.3.1 Evaluation metrics

Metrics are quantifiable measures useful to evaluate the quality of a technique, crucial in determining its adoption on a commercial scale. Since malware detection is a classification problem, the evaluation of the detectors relies on the standard

classification metrics, which can be clustered as **performance** and **efficiency metrics**. Performance is the degree to which a system or component accomplishes its designated functions within given constraints, i.e., correctly detects the malware. Efficiency is the degree to which a system or component performs its specified functions with minimum consumption of resources, i.e., power consumed in malware detection [80].

The performance metrics for classification and malware detection start from the **confusion matrix**, depicted in Table 2.3. Considering the actual label of a sample (benign or malware) and the predicted value (again, benign or malware), the results are organized in four groups: True Positives (TPs) represent samples where the model correctly predicts malware presence, True Negatives (TNs) indicate correct predictions of benign presence. In contrast, False Positives (FPs) and False Negatives (FNs) denote incorrect predictions of malware presence or benign presence, respectively.

Table 2.3: Confusion matrix. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

	Predicted negative	**Predicted positive**
Actual negative	True negatives (TNs)	False positives (FPs)
Actual positive	False negatives (FNs)	True positives (TPs)

The confusion matrix allows for the definition of more descriptive performance metrics, with the most common presented in Table 2.4 [81, 82, 83]. The **accuracy** is one of the most widely used metric, it summarizes the overall correctness classification by expressing the number of correct predictions. The **precision** is proper when correct malware prediction is crucial, it provides an accurate measure of the correct malware predictions among all malware predictions. When it is important to ensure that no malware passes unnoticed, the **True Positive Rate (TPR)** (also known as **recall** or **sensitivity**) weighs the correct malware predictions against all malware samples. It has two counterparts: (i) the **False Positive Rate (FPR)**, representing the probability of a correct malware prediction being missed, and (ii) the **specificity**, indicating the probability of a benign being correctly predicted. The **F1-score** balances precision and recall, representing their harmonic mean. The **Receiver Operating Characteristic (ROC)** curve offers a visual performance perspective. It plots the TPR against the FPR, enabling a visual comparison and the inspection of the **Area Under the Curve (AUC)**. The larger the AUC, the better the classifier. AUC is closely related to the robustness of the classifier, indicating how effectively the classifier distinguishes between malware and benign applications.

Efficiency is the other group of metrics employed with the evaluation of classifiers, being related to the physical resources used to perform malware detection.

Table 2.4: Most common metrics for performance evaluation of classification and malware detection. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

Metric	Expression
Accuracy (A)	$A = \frac{TP + TN}{TP + TN + FP + FN}$
Precision (P)	$P = \frac{TP}{TP + FP}$
True Positive Rate (TPR)	$TPR = \frac{TP}{TP + FN}$
False Positive Rate (FPR)	$FPR = \frac{FN}{FN + TP}$
Specificity (S)	$S = \frac{TN}{TN + FP}$
F1-score (F1)	$F1 = \frac{2 \times (P \times R)}{(P + R)}$
Receiver Operating Characteristic (ROC)	$ROC = 1 - S = \frac{FP}{FP + TN}$
Area Under the Curve (AUC)	$AUC = \int_0^1 R(FPR)\, dFPR$

Various metrics can be used to assess efficiency [84], but latency, power consumption, and hardware cost are the primary focus of malicious code detection:

- **Latency** is the time delay between gathering all samples analyzed by the malware detector and concluding its detection. A low latency is crucial for run-time detection, especially when dealing with malware that acts in a short interval of time;
- **Power consumption** indicates the amount of energy the detector consumes per unit of time. Two factors primarily impact the power consumption: the hardware that implements and the detection algorithm (those with higher computing processing tend to consume more);
- **Hardware cost** refers to the monetary expense of building the detection system, which is a critical factor in determining its financial viability for industry and research applications. The main parameter to evaluate the hardware cost is the chip area (typically reported in square millimeters) in conjunction with the process technology (e.g., 45 nm). Sometimes, the amount of memory is also considered in the hardware cost evaluation.

2.3.2 Software-based malware detection

Software-based detection relies on software tools and techniques to identify and analyze the presence of malware. It encompasses various approaches and analysis techniques (see Figure 2.3):

- **Signature-based** is widely used within commercial antivirus, does not allow zero-day detection [10] and polymorphic/metamorphic malware detection. The signature is a unique malware feature extracted from structural properties (e.g., code sequences and hash values) or run-time properties (e.g., processor and memory information, kernel usage, file system activities, and network communications) [85]. The detection operates as follows: features extracted from the executable generate a signature stored in a signature database. When the system needs to classify a potential threat, the detector extracts the related features and computes the corresponding signature. This signature is compared against those in the database. If a match is found, the potential threat is marked as malware;

- **Behavior-based** allows zero-day detection and polymorphic/metamorphic malware detection, which are challenging for the signature-based approach. It is performed by observing dynamic characteristics from run-time executions of programs [86]. The dynamic characteristics might include processor and memory information, kernel usage (system calls), file system activities, and network communications. They are extracted with monitoring tools and analyzed by ML detectors to distinguish benign applications from malware;

- **Heuristic-based** can perform zero-day detection and is effective against polymorphic/metamorphic malware, similar to the behavior-based approach. The heuristic-based approach relies on experiences and techniques, including patterns, behaviors, rules, and ML. The process involves two phases: in the first, the detector is trained with benign applications and malware data to identify relevant characteristics; later, the monitoring or detection occurs, the trained detector intelligently assesses new samples to make decisions [87];

- **Deep Learning** proves highly effective for zero-day detection and polymorphic/metamorphic malware. The approach falls under the umbrella of ML algorithms, using deep learning techniques to extract more advanced features from raw input. The Feature Extraction (FE) combines elements from previous approaches, making it a novel approach, which facilitates context adaptation and model updates [10].

Three analysis techniques may be employed with these detection approaches mentioned [85]: (i) **static analysis**, using structural properties of the program (e.g., code sequences, hash values and metadata) without executing it; (ii) **dynamic analysis**, extracting the necessary data during or after program execution, leveraging run-time information; (iii) **hybrid analysis**, combining the two previous. Selecting one of these approaches also affects the expected detection latency. While static analysis aims to identify threats before the malicious program is executed, the other two methods may require an entire execution before detection.

2.3.3 Hardware-based malware detection

HMD are more detection approaches that address the performance and efficiency of traditional software-based detection with distinct advantages. They involve dynamically analyzing architecture and microarchitecture events in a CPU using ML algorithms to distinguish between benign applications and malware. The intuition that malware can be detected based on architecture and microarchitecture hardware events stems from the idea that programs have phase behaviors [88, 89]. The phases vary significantly between programs and reflect patterns in hardware events (see Figure 2.4). This enables differentiating programs based on their time-behavioral hardware event patterns, facilitating the distinction between benign and malicious applications. The architecture and microarchitecture events are monitored by Hardware Performance Counters (HPCs) present in modern processors (see Section 3.1 for a detailed discussion), and various ML techniques can be applied to the HPCs collected [9]. The HMD approaches arose in the early 2010s. In 2011, Malone et al. [90] demonstrated the feasibility of detecting program code modifications based on the deviation of hardware events. In 2013, Demme et al. [91] proved the feasibility of detecting Android malware and Linux rootkits using values of hardware events analyzed by a ML classifier.

The capability for **runtime detection** is one of the main advantages of HMD. The analysis leverages real-time hardware collected data, and with fast ML classification, a few milliseconds are required to identify the threats. This translates to low latency, enabling runtime detection [92, 93, 94]. Another advantage of HMD is the capability for **zero-day detection and stealthy malware detection**. Unlike static analysis employed by most software-based antivirus solutions, the dynamic analysis performed by HMD provides deeper insights into how applications operate, suitable for detecting more elaborated threats [91]. Moreover, HMD is also **resilient against subverting the protection**. While software-based approaches employ software and are susceptible to bugs or oversights in the underlying system, HMD approaches with secure hardware significantly reduce the possibility of turning off the protection [91, 95]. Still, HMD imposes **low-performance overhead** compared to software-based approaches, whose dynamic analysis necessitates sophisticated computation, often at the expense of significant performance overhead. The increasing software size further complicates dynamic software analysis [91]. Conversely, in the HMD, understanding software behavior provided by architectural and microarchitectural events simplifies the analysis, reducing computational processing efforts and the **cost** of detection [91, 96].

Despite the advantages of HMD approaches, they also have some weaknesses. Their **effectiveness** remains open to discussion. On the positive side, [91, 95] reported accuracy consistently exceeding 80%, deeming it effective. However, after detailed experiments of their own, [97, 98] argued that detection capabilities reported by other works often stem from tiny sample sizes and experimental setups

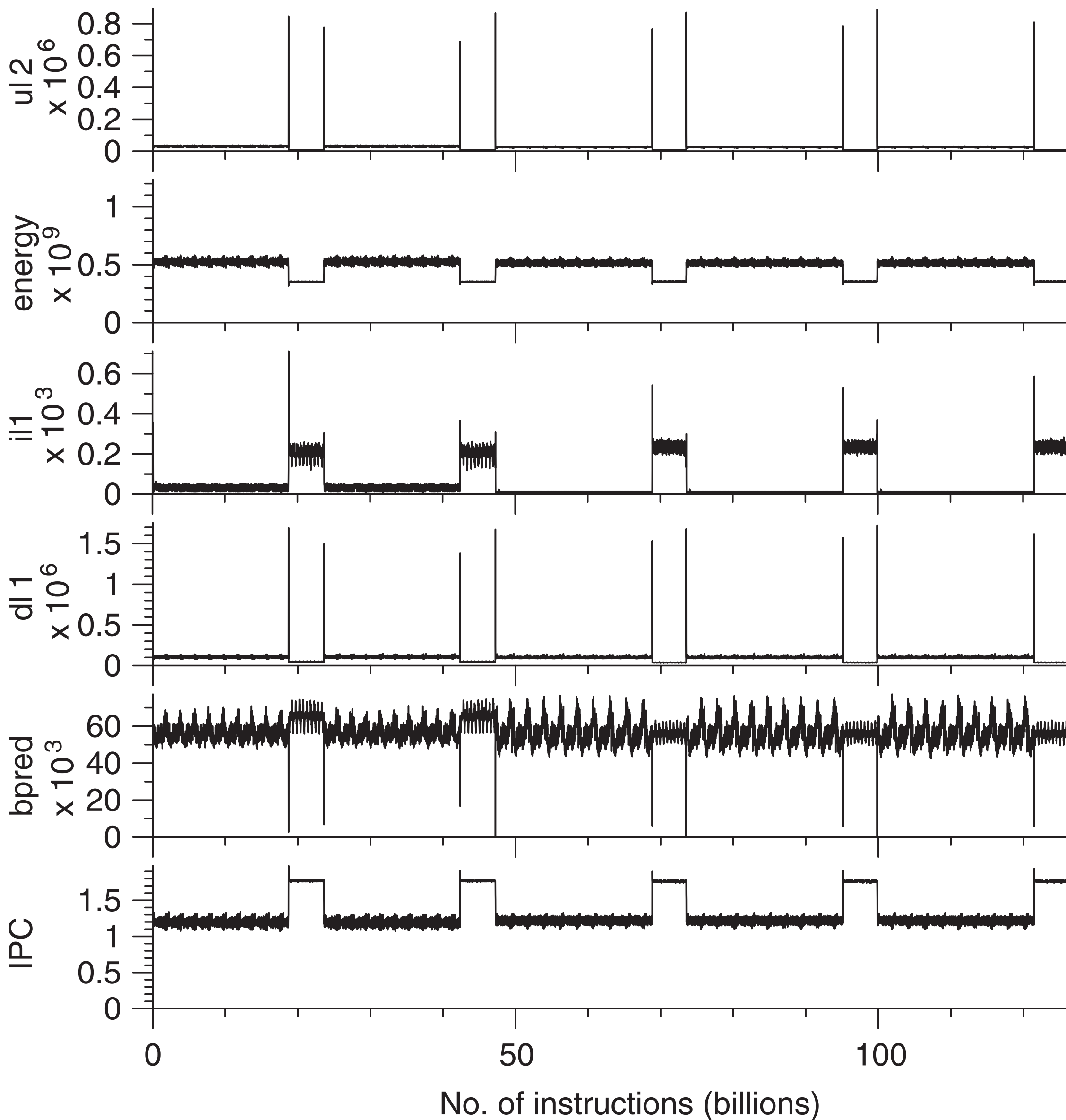


Figure 2.4: Example of patterns in architectural and micro-architectural hardware events reflecting the program phases, enabling distinguishing between benign and malicious applications [88].

that favor the detection mechanism unrealistically. They added that even accurate, an 80% accuracy is insufficient in cases with thousands of executables, risking many benign applications being misclassified as malware. They also questioned the causal link between low-level architectural and microarchitectural events and high-level software behavior. Finally, they illustrated the inability of a hardware-based detector to distinguish ransomware embedded in a benign application like Notepad++.

Recently, [99] acknowledged the absence of a perfect malware detector and argued that HMD is only effective for specific types of malware. In particular, they claim that the approach is effective with attacks exploiting architectural side-effects and attacks that can be detected via architectural anomalies they cause. As examples they cite: RowHammer [100, 101], detected via excessive cache flushes [102]; ROP attacks [65], identified by an abundance of instruction misses [103]; and DirtyCoW [104], detectable through heightened paging activity. The authors also emphasized the necessity for a maliciousness theory to enhance the understanding of malware threats and assess proposed defenses.

Inconsistencies in HPCs also contribute to the weakness of HMD approaches. They have been studied since their use in the past for other purposes, like optimization, performance analysis, safety, and security. Weaver and McKee [105] performed experiments to observe inter-machine and intra-machine variations in HPCs. Zaparanuks et al. [106] compared true event counts with measurements performed by different extraction tools and processors. Weaver et al. [107] performed experiments with several processors to identify the causes for the **non-determinism** and **overcounting** of HPCs. Non-determinism refers to identical execution that returns different values. In X86 processors, hardware interrupts and page faults were observed as their cause. Overcounting refers to instructions being counted multiple times. In X86 processors, the causes were identified as the overflow of X87 top-of-stack pointers, specific conditions for using the floating point unit, and the measurement of microcoded events rather than retired events.

Das et al. [108] highlighted some of these HPC challenges in security. Besides the non-determinism and overcounting, they added to the causes list the idea of **external sources** and **variations in the implementation of tools** that obtain the HPC. External sources refer to variations in the run-time environment (operating system activity, scheduling of programs, memory layout and pressure, and multi-processor interactions). Recent studies address HPC discrepancies, propose methodologies, analyze resilience, and compare HPCs in various machines [109, 110, 111, 112].

Another concern in HMD is the **device dependency**. The characteristics of PMUs (discussed in Section 3.1) are determined by the device architecture and manufacturer, and detectors should be designed to align with these specifications. For example, PMUs may track different events, may have different number of HPCs, and discrepancies in instruction counting methods are possible [105]. These factors underscore the need for malware detection applications to abstract software from the hardware level. Moreover, the ML classifiers built on top of the HPC frequently lead to techniques that increase the complexity of such algorithms, like ensemble learning and time series or even Deep Neural Networks (DNNs) [9].

Some countermeasures can be deployed to minimize the inconsistencies from HPCs [105, 108]. They entail per-process filtering of events (applied by saving and restoring the counter values at context switches), proper interrupt handling, and

minimizing the impact of non-deterministic events. All works generally recommend careful use of HPCs. The evolution and improvement of the CPU PMUs also reduce this issue.

Chapter 3

Hardware-based malware detection approaches

Adapted, with permission, from Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo, "A Survey on Hardware-Based Malware Detection Approaches," in IEEE Access, 2024, doi: 10.1109/ACCESS.2024.3388716. Licensed under CC-BY.

Hardware-based malware detection demonstrated some advantages, like the ability for runtime detection, capability for zero-day detection and stealthy malware detection, resilience against subverting the protection, low-performance overhead in comparison with software-based approaches and reduced cost of detection. Conversely, it also offers some weaknesses that require additional effort to enable its future use on a large scale, as seen in Chapter 2.

This chapter delves into the various critical points of HMD. One of them is related to the ISA, an abstract model that acts as an interface between the hardware and the software, specifying how the CPU is controlled by the software. The ISA establishes the basis for the instructions that the CPU can execute, the organization and availability of registers, memory management mechanisms, supported data types, addressing modes, and other key points that together define the CPUs microarchitecture. The dominant CPU manufacturer/designer companies use specific ISAs and CPU microarchitectures. Their hardware events are counted by dedicated PMUs, silicon-based hardware units that are fundamental building blocks for the HMD approaches. Table 3.1 presents the dominant CPU manufacturer/design companies, their ISAs, and end-user markets.

Section 3.1 discusses the starting point for the approaches, the hardware events, and the PMUs. Section 3.2 focuses on the implementation, using a generic detection framework to explore the possibilities methodically. Section 3.3 overviews the main reference studies in the field, and summarizes the performance and efficiency of the

Table 3.1: Dominant CPU manufacturer/design companies, their ISAs and end-user markets.

Manufacturer	Instruction Set Architecture	End-user markets
Intel	X86 [113]. It is a well-established family of ISAs based on the CISC principles. Intel introduced it in 1978 with the 8086 microprocessor, which used memory segmentation to address more memory than can be covered by a plain 16-bit address. The term X86 is due to the names of several successors of 8086, including the 80186, 80286, 80386, and 80486, colloquially called 186, 286, 386 and 486	Servers, desktops and laptops
AMD	X86 [113]	Servers, desktops and laptops
ARM	Proprietary. ARM develops and licenses its well-established family of ISA, which are based on the RISC principles	Battery-powered devices (smartphones, tablets, laptops) and embedded systems; Less used in desktops
Several	RISC-V [11]. It is an emerging platform that is attracting attention. The European governing bodies are making a strong effort to invest in it [114] as a solution to achieve technological sovereignty. It is based on established RISC principles, designed to offer simplicity, modularity, and extensibility, and provided under royalty-free open-source licenses. This licensing approach brings benefits over proprietary processor architectures, including the potential for customization and lower licensing costs [115]	Battery-powered devices (smartphones, tablets, laptops) and embedded systems; Servers, desktops and laptops; Academia, research and strategic use

approaches' state-of-the-art. Section 3.4 discusses some ML techniques currently employed to improve the performance of HMD.

3.1 Performance Monitoring Units

PMUs are designed to count events from the CPU architecture and microarchitecture. They are composed of some HPCs, each one with an event detector and an associated counter (Figure 3.1). Multi-core processors have a separate PMU for each core, each one configured individually. In 2002, Sprunt [116] published a seminal paper on the basics of PMUs. These units were developed to enable programmers to tune algorithms and code performance without the necessity of instrumenting the source code. The proven advantages from PMUs utilization and their spreading availability among different devices have led to their leverage for safety and security purposes [117, 118, 119, 120].

The sophistication of modern processors results in several events to monitor. Complex devices like high-end processors have hundreds of events to monitor. However, to minimize the PMU hardware complexity, only a few HPCs are available, typically ranging from 2 to 8 in high-end processors, thus limiting the number of parallel events that can be monitored. Programmers should choose the events configuring the PMU, respecting the HPCs constraint.

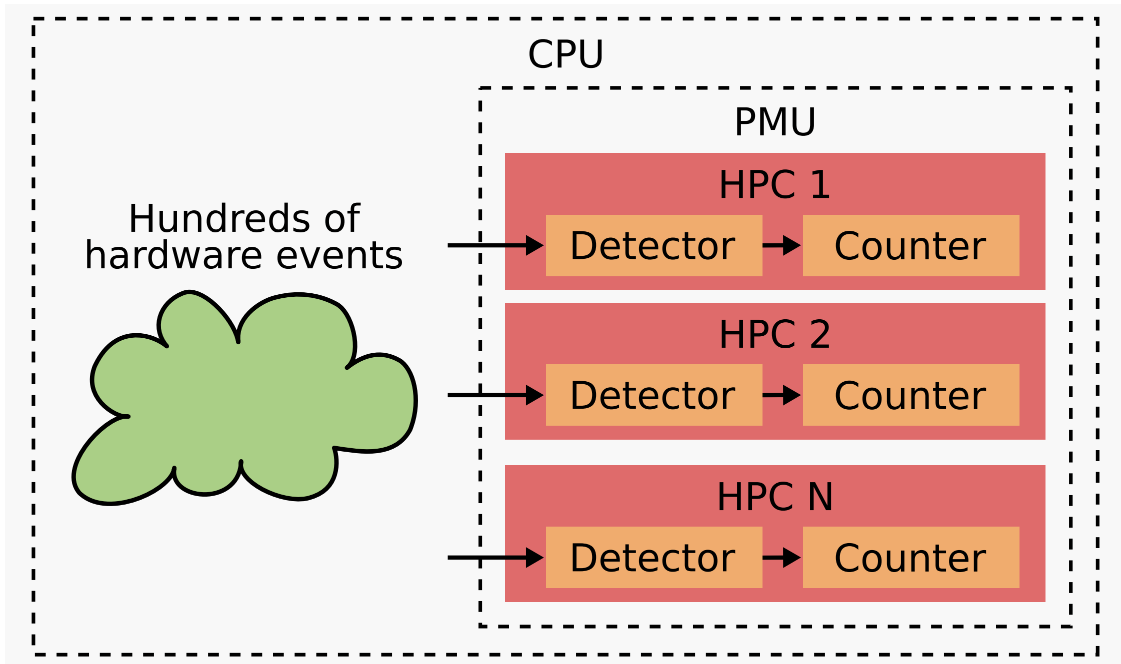


Figure 3.1: Performance Monitoring Units. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

The academic literature presents many examples of stand-alone PMUs, targeting efficient and accurate units [121, 122, 123, 124]. The type of events monitored varies by CPU microarchitecture but typically may be clustered with relation to the CPU pipeline, branch predictor, caches (load and store operations), floating-point operations, clock blocks, and others.

- **Pipeline**: instruction fetch, decode, issue, dispatch, execute, retirement, and stall;
- **Branch predictor**: branch execution, conditional branches taken and not taken, and mispredicted conditional branches;
- **Caches (load and store operations)**: accesses to the L1 data cache (hits, misses, and fills); accesses to the L2 and L3 cache; Translation Lookaside Buffer (TLB) (accesses, hits, and misses) and page table walks;
- **Floating-point operations**;
- **Clock blocks**: frequency and elapsed core clock ticks;
- **Others**: hardware interrupts, temperature, etc.

Understanding these events is often not intuitive and requires a background in computer architecture and knowledge of the specific microarchitecture analyzed. The Section 3.1.1 addresses these concepts.

3.1.1 CPU microarchitecture concepts

Reviewing concepts related to the CPU pipeline, branch predictor, and caches enables the understanding of most of the events possibly monitored by the HPCs. Several references in computer architecture may be used for this study [125, 126, 127, 128].

The anatomy of a pipeline mainly reveals the CPU major functional units and how they work. Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Figure 3.2 shows the pipeline stages for the Microprocessor without Interlocked Pipelined Stages (MIPS) microarchitecture [125], which is a typical example for educational purposes. The major functional units are the Program Counter (PC), the instruction memory, the registers, the Arithmetic Logic Unit (ALU), and the data memory. The CPU works in five pipeline stages, performing a specific part of the instruction execution process: (i) all instructions start by the **instruction fetch**, in which the CPU fetches the instruction from instruction memory based on the PC; (ii) after the instruction is fetched, the **instruction decode** is performed by reading the operands used by the instruction in the registers block; (iii) once the instruction is decoded, the CPU can perform the **instruction execute**, that may be compute a memory address (for a load or store), or compute an arithmetic result (for an integer arithmetic-logical instruction) or calculate and compare the target address (in the case of conditional branches); (iv) after the instruction execution, if the instruction is a load or store, **memory access** is performed to read/write in the data memory; (v) the last stage is the **write back**, which consists in writing to the register the result of the ALU operation or the data memory read (such as an arithmetic instruction).

Sometimes, a particular pipeline stage becomes a bottleneck. Modern CPUs alleviates this problem by using multiple functional units, as proposed by Sohi [129] and shown in Figure 3.3. The multiple functional units operate in parallel, solving the bottleneck in the execution stages. The decode stage has an instruction queue, and the **instruction dispatch** is the process of taking instructions from this queue. The **instruction issue** is the process of sending these instructions for execution in the various pipeline stages. The **instruction retirement** refers to the process of finalizing the execution of an instruction and writing the final results back to the registers and memory. An **instruction stall** occurs when the CPU must temporarily halt the execution of instructions due to certain constraints or dependencies. The pipeline, which typically allows multiple instructions to be processed simultaneously in different stages, is stalled to accommodate this issue. Stalls reduce the CPU performance due to the introduction of idle cycles.

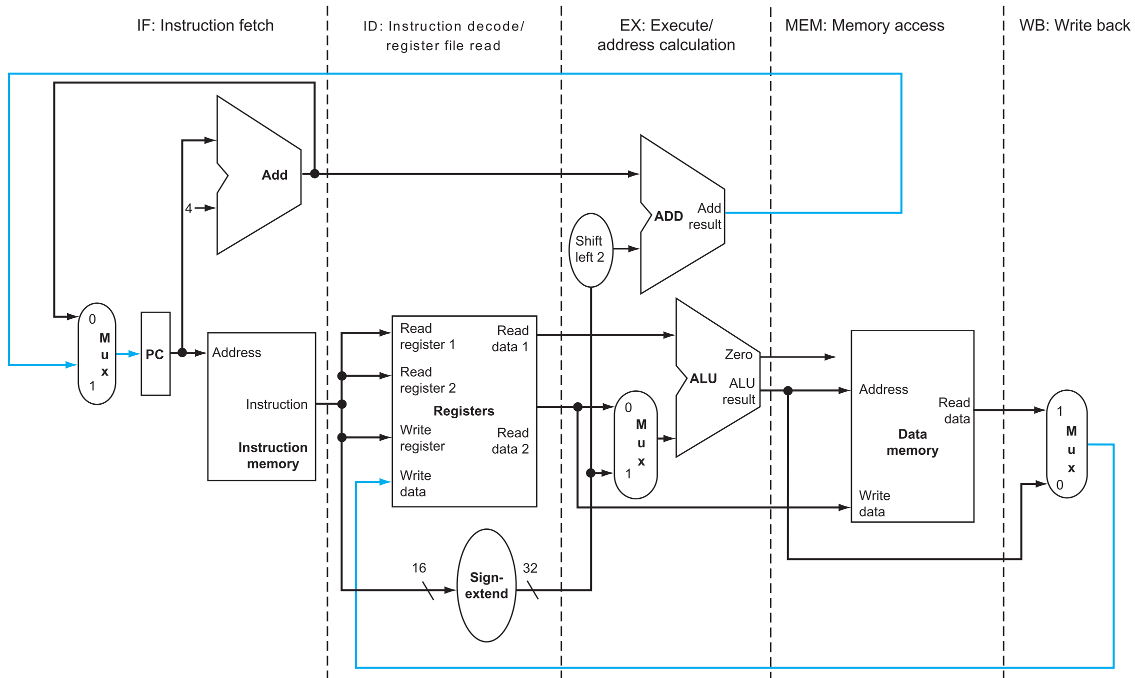


Figure 3.2: MIPS pipeline stages [125].

Another important concept in CPU microarchitecture is of **branch execution**. It refers to the execution of an instruction that may cause a jump to another part of the program, altering the execution flow. Branches are essential for implementing loops, conditional statements, function calls, and returns. They can be unconditional branches (like a `goto` statement) or conditional branches (that require the comparison of two values and allow a jump in the program based on the outcome of the comparison). **Conditional branches taken** are when the branch condition is satisfied and a jump is performed to deviate the program flow. **Conditional branches not taken** are when the branch condition is false, and the program follows the sequential branch. Branches offer a challenge for the pipeline performance because the CPU does not know which path to take until the branch condition is evaluated, possibly generating stalls that impact the performance. To address this challenge, modern CPUs use branch predictors. They are a hardware mechanism that predicts the outcome of a branch instruction before the condition is evaluated, keeping the pipeline filled with instructions and avoiding unnecessary stalls. The implementation of the predictor may employ different techniques. The **mispredicted conditional branches** occur when the CPU predicts the wrong outcome for a branch, resulting in a performance penalty.

Turning to memory, one of the critical aspects of computer architecture, responsible for load and store instructions. Due to the principle of temporal and spatial locality of data [125, 126, 127, 128], the memory of computers is implemented as

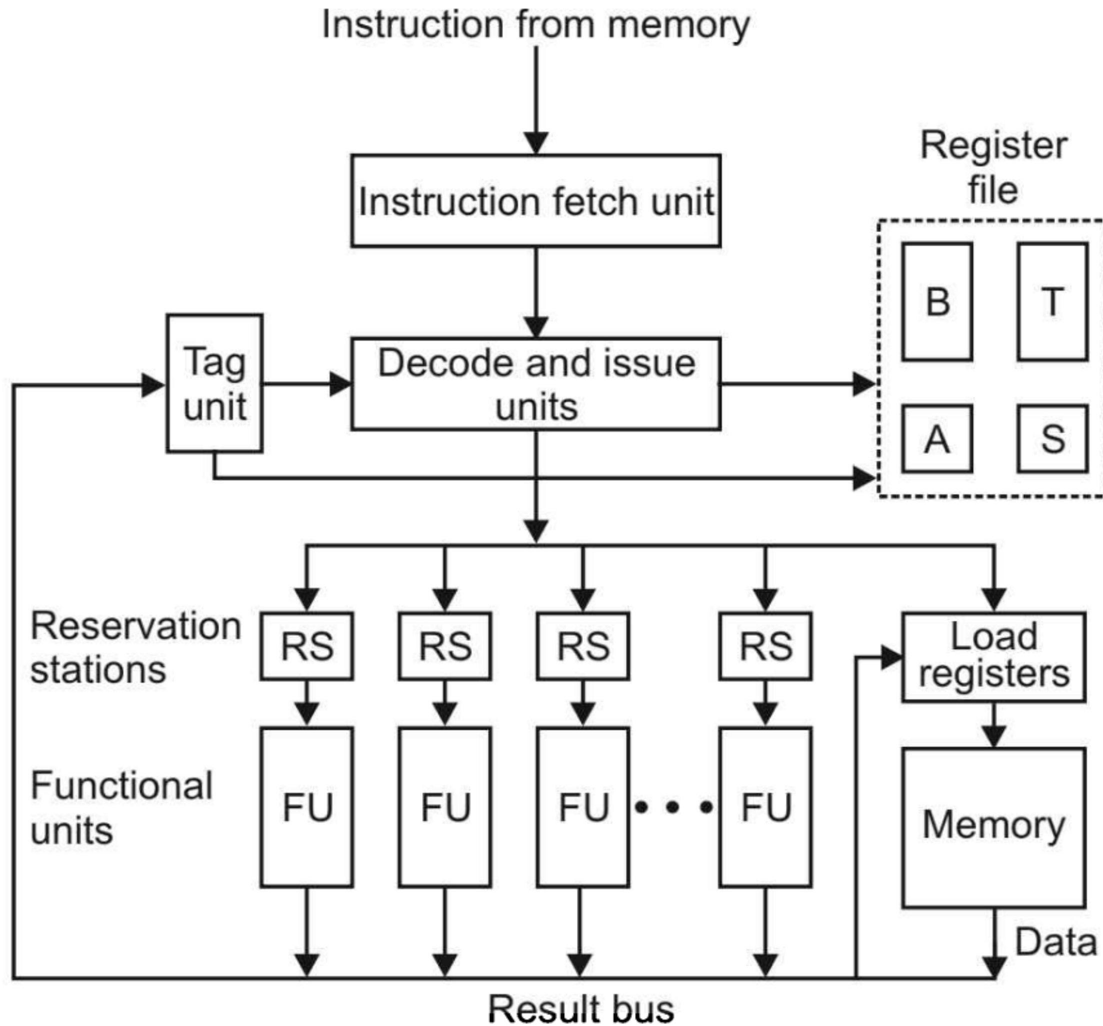


Figure 3.3: Pipelined CPU with multiple functional units [129].

a memory hierarchy. The memory hierarchy consists of multiple levels of memory with different speeds and sizes. As the distance from the CPU increases, the size of the memories and the access time increase. The data is similarly hierarchical: a level closer to the CPU is generally a subset of any level further away, and all the data is stored at the level further from the CPU. On the hierarchy, data is copied between only two adjacent levels at a time. By convention, the level closer to the CPU is called the L1 cache, the underlying L2 cache, and so on. If the data requested by the CPU is found in the L1 cache, this is called **L1 cache hit**. The request is called **L1 cache miss** if the data is not found. In the case of an L1 cache miss, the next lower level in the memory hierarchy (L2 cache) is then accessed to retrieve the data requested by the CPU, what is called **L1 cache fill**. The analog process happens at the next memory level (L2, L3, and beyond).

Sophisticated techniques to deal with memory were developed. One of these techniques is the virtual memory (Figure 3.4), created to allow many programs to share a computer's physical memory. Each program is compiled into its own address space, a separate range of memory locations accessible only to this program. The virtual memory system implements the translation of the program's address space (virtual addresses) to physical addresses. The virtual and physical memories are broken into pages, so a virtual page is mapped to a physical page. A page table is used to map/translate the page number from the virtual address to the corresponding physical page number.

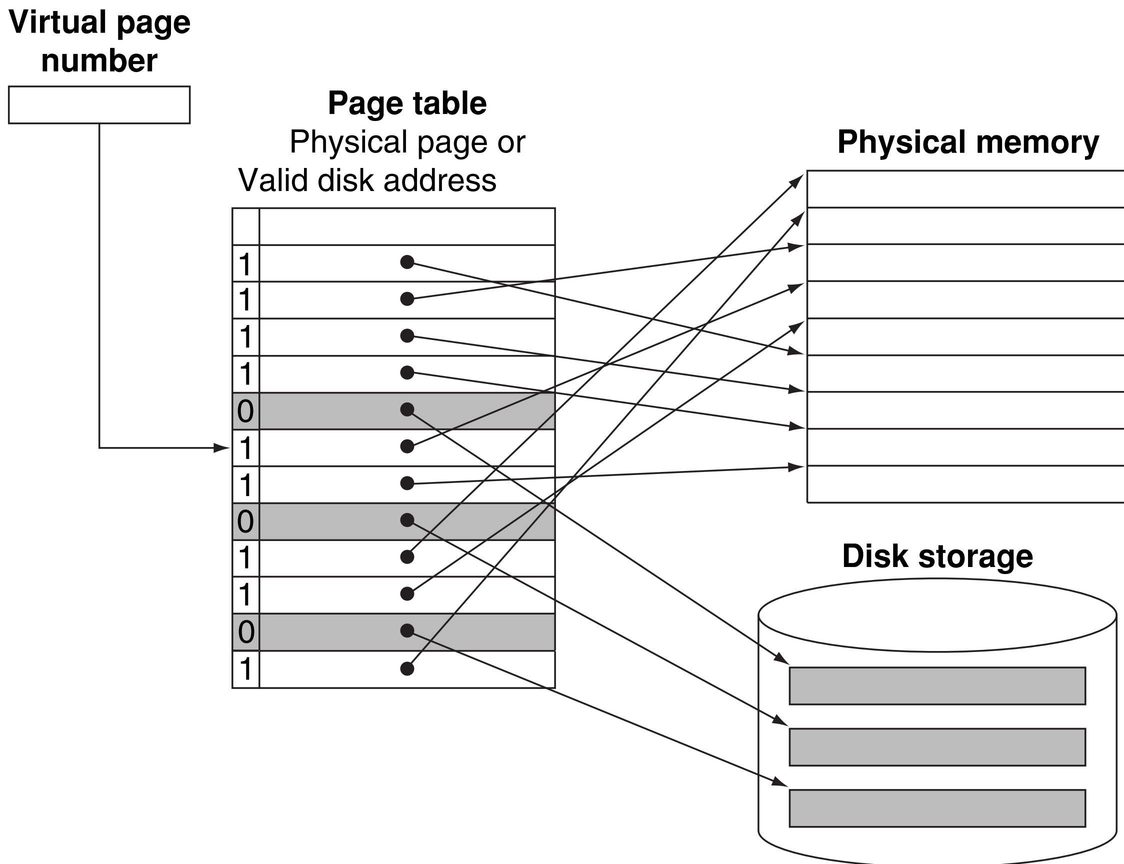


Figure 3.4: A virtual memory system [125].

Since the page tables are stored in physical memory, every memory access by a program can take at least one memory access to obtain the physical address and a second access to get the data. Modern processors include a cache that keeps track of recently used address mappings to avoid access to the page table, called TLB. If the CPU requests a translation to the TLB, we have a **TLB access**. If the data is found in the TLB, we have a **TLB hit**. Otherwise, we have a **TLB miss**. In the case of a TLB miss, a **page table walk** retrieves the required mapping from the

page table stored in memory, and the result is cached in the TLB for future use. Page table walks are a crucial component of virtual memory systems, but they are expensive due to the latency of accessing physical memory.

3.1.2 Intel PMU

Intel CPUs implement the X86 family of ISAs. The PMU counters and configuration registers are called Model-specific Registers (MSRs). They are divided into **event select MSRs** (control the PMU operation) and **counter MSRs** (contain the event or duration counts for the selected events). In most Intel processors, each CPU core has four fully programmable counter MSRs and three fixed function counter MSRs. Fixed counter MSRs usually monitor core clocks, reference clocks and instructions retired [130]. [131, 132] contain the events for all Intel microarchitectures, including the most recent Intel CPU microarchitectures for performance (Golden Cove and Raptor Cove). The events are classified into core events (counted from within a logical CPU core), uncore events (related to logic outside of the CPU core), and offcore (related to events that involve multiple CPU cores). Modern hybrid/heterogeneous processors, containing different kinds of CPUs dedicated to performance and efficiency, also have P-core (performance) and E-core (efficiency) events.

3.1.3 AMD PMU

Like Intel, AMD CPUs implement the X86 family of ISAs. An introductory reference on AMD PMU is in [133]. The PMU counters and configuration registers are also called MSRs. They are divided into three types: **time-stamp counter register** (used to count CPU clock cycles), **PerfEvtSeln registers** (control the PMU operation), and **PerfCtrn registers** (contain the event or duration counts for the selected events). All AMD CPUs support the base set of four fully programmable counters plus some extended MSRs to monitor Northbridge and L2 cache events (which vary by specific CPU). [134] details the PMU for the Zen 5, the AMD CPU microarchitecture released in 2024. The hardware events are divided into floating-point events, load and store events, instruction cache and branch prediction events, data exception events, execution and store conditional operations, and L2 and L3 cache events.

3.1.4 ARM PMU

ARM CPUs have proprietary ISAs. Their CPUs for high-performance computing (Artificial Intelligence (AI), personal computers, mobile, gaming, and enterprise) belong to architectures that ARM clusters like "Application-profile." It comprises three microarchitecture families: Cortex-A, Neoverse, and Cortex-X. The

PMU for these CPUs is detailed in [135]. The PMU basic form contains one **cycle counter**, one optional **instruction counter**, and **up to 31 programmable event counters**. The number of event counters is defined by the third parties using ARM CPU intellectual properties. ARM defines a list of **common events** across many architectures and microarchitectures, which may be selected to be counted. The events related to microarchitecture characteristics of the implementation not covered by common events belong to the **implementation-defined events**.

3.1.5 RISC-V PMU

In RISC-V ISA, the privileged specification [136] gives some baselines for the PMU implementation. Version 1.7 was the first RISC-V attempt to specify the PMU, and now the current version (1.13) maintains the exact specification since version 1.11. The PMU in this current specification contains one cycle counter (**mcycle**), one instruction retired counter (**minstret**), and 29 additional programmable counters (**mhpmcounter3-mhpmcounter31**), controlled by 29 registers (**mhpmevent3-mhpmevent31**). The events to monitor are not defined by RISC-V specification, being dependent on the implementation. A counter-enable register (**mcounteren**) controls the availability of the counters to the next-lower privileged mode. It only controls accessibility and does not affect the underlying counters, which continue to increment even when not accessible. A counter-inhibit register (**mcountinhibit**) controls which counters increment.

As can be seen, the PMU in the RISC-V specification is still reduced. It does not specify the events to monitor nor a mechanism to generate an interrupt when a counter overflows. More detailed PMU information should be searched for the targeted implementation of RISC-V.

3.2 Hardware-based detection framework

The implementation of HMD approaches can be guided by the generic framework presented in Figure 3.5. It leverages the existing PMU within the CPUs and consists of two main components: (i) data collection and preprocessing and (ii) malware detection. The primary goal is to differentiate applications between benign and malicious. This section provides a detailed overview of the implementation process.

Data collection and preprocessing involves FE and Feature Selection (FS) [137, 138, 139, 140]. FE captures and stores HPCs in a vector space. FS focuses on selecting a subset that efficiently describes HPCs while reducing noise and irrelevant variables, ensuring optimal prediction results. FE can occur in the time or event domain [116]. In the time domain, the execution of the application is periodically interrupted to store the HPC values. Conversely, in the event domain, the

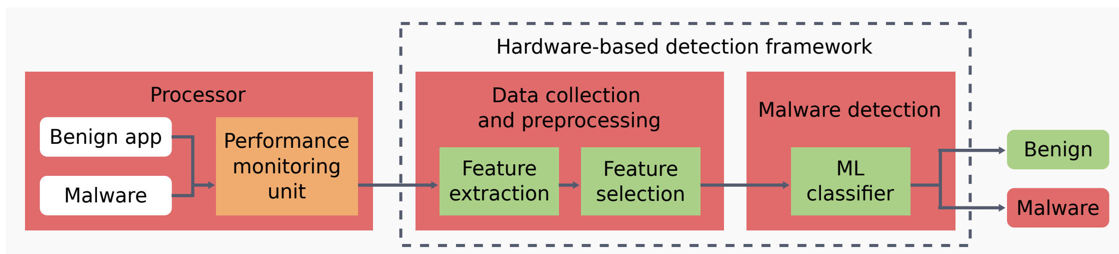


Figure 3.5: Generic hardware-based detection framework. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

interruption of the application's execution is triggered after several specific events or instructions.

There are four strategies to perform the **FE** of HPCs: (i) instrument the source code with the employment of a library, like `PAPI` [141]; (ii) develop of a proprietary kernel module or driver, as performed in [95]; (iii) use of an available utility that performs tasks mainly in the OS kernel, like `PERF` [142]; and (iv) use of a simulator to model the CPU as it executes the application, like `gem5` [143] and `GVSoC` [144].

When performing FE, the sampling strategy is crucial. In the time-based domain, parameters like period, frequency, or number of cycles determine when HPCs are sampled. No fixed ideal value exists, varying based on the experiment and goal. HMD experiments typically use periods in the order of milliseconds or seconds. Striking a balance between low and high sampling values is essential, considering the trade-off between computational processing, data quantity, and system effects. In the event-based domain, sampling depends on the number of event or instruction occurrences. The sampling definitions are also influenced by the FE strategy. A proprietary kernel module or driver gives great freedom, allows programmers to choose between time-based or event-based domains, set parameters for sampling triggering, and specify values. However, limitations may exist when libraries and available utilities are used due to the configurations accepted.

FS offers multiple benefits, like avoiding the curse of dimensionality [145], reducing the computational effort, enhancing data interpretability, improving classification performance and robustness, and handling noisy and correlated features. The curse of dimensionality is the reduction of detection rates when working with high-dimensional data. The redundant dimensions turn into sparse and less meaningful data, making classifiers struggle to generalize effectively. Among FS methods, filter-based dominates the HMD field, focusing on the relevancy of features. It applies statistical techniques to assign scores to each feature, ranking them accordingly and selecting features based on a predefined threshold. This method is valued for simplicity and practical application success. Most common filter-based algorithms include Principal Component Analysis (PCA) (used by [97, 93, 146, 147]), Fisher Score [148] (used by [95, 149]), Pearson Correlation Coefficient [150] (used by [94,

92, 93, 146, 147]) and Information Gain (Mutual Information) [151] (used by [152, 153]). The Scikit-learn [154] library for the Python and Weka workbench [155] are tools frequently used in the HMD field for FS.

Some studies (for example, [94, 90, 156]) also perform a preliminary manual FS before data collection since the number of events that can be potentially monitored exceeds the available HPCs. In this case, the selection is guided by previous studies and the knowledge of the CPU architecture and microarchitecture.

The second principal component of the HMD framework is **malware detection**, in which ML classifiers play a crucial role. In hardware-based detection, supervised and unsupervised learning classifiers are often employed. While for supervised detectors, both benign and malignant annotated samples are necessary, in unsupervised detectors, the classifier is trained with samples without labels (benign or malign). Unsupervised detection has two exciting advantages: (i) it does not require malignant samples for training, and (ii) it can detect zero-day malware. Conversely, algorithms for unsupervised detection may be more complex.

Like in FS, the Scikit-learn [154] library for Python and Weka workbench [155] have tools frequently used in the HMD field for implementing ML classification. The Weka workbench clusters its available classification algorithms in: linear regression (LinearRegression and SimpleLinearRegression), logistic regression (Logistic and SimpleLogistic), Bayesian network (BayesNet and NaiveBayes), decision trees (J48 and REPTree), rule-based (JRIP, OneR, and PART), Artificial Neural Network (ANN) (MultiLayerPerceptron), K-Nearest Neighbors (KNN) (IBk), ensemble learning (AdaBoostM1, Bagging and RandomForest) and Support Vector Machines (SVM) (SMO). The algorithms in parentheses refer to specific Weka implementations. Further details can be found in the documentation [155].

An important consideration in hardware-based malware detection is related to the number of events monitored and their impact on runtime applicability. More events better characterize the application and improve the detector performance but require multiple application executions if exceeding available HPCs (like in [91, 152, 157]). This trade-off is further addressed in ML solutions discussed in Section 3.4.

3.3 Hardware-based detection assessment

Table 3.2 contains reference studies in HMD, outlining the scenarios regarding targets, classifiers tested, and system characteristics. The Weka implementation is the reference for the classifiers. The selection of these studies was based on the following criteria: papers with the most citations, cited by these, and most recent publications. Of all the contributions reported, [158] deserves a highlight. It showcases real scenarios: DARPA Rapid Attack Detection, Isolation and Characterization Systems (RADICS) [159], Intel Threat Detection Technology (TDT)

[160], and Microsoft Defender for Endpoint [161]. These cases are a tangible exploitation of HMD into actual products. Still, using a single type of classifier (i.e., SVM) leaves room for research and improvements. The remainder of this section analyzes the performance and efficiency of these reference studies in HMD.

3.3.1 Performance

Table 3.3 summarizes best-case performances from reference studies in Table 3.2. The values are all normalized between 0 and 1 (ideal performance). The "HPCs" column refers to the number of hardware events the classifiers consider. The "classifier" column denotes the algorithm associated with the best result. The evaluation metrics were directly sourced from the studies' text whenever feasible, with manual extraction from ROC curves only when necessary. The values in Table 3.3 underscore the efficacy of HMD in supporting malware detection and highlight the overall quality of the findings. In most cases, performance metrics exceed 85%, showcasing the potential of the approaches but also leaving room for improvements.

Concerning specifically the classifiers, Patel et al. [94] analyzed the performance and efficiency of ML classifiers when applied to HMD. They are particularly interesting since most of the current studies in these approaches rely on ML classifiers. Table 3.4 summarizes the findings. The authors thoroughly analyze eleven algorithms based on Weka workbench [155] implementations. The dataset used for training and testing the algorithms was generated in the following scenario: Intel Haswell Core i5-4590 processor running Ubuntu 14.04 with Linux kernel 4.4; feature extraction using PERF tool in intervals of 10 ms; malware from the VirusTotal dataset [163] and benign applications from Mibench benchmark suite [165], Linux system programs, browsers, text editors and word processor. The study covers different numbers of hardware events (i.e., 32, 8, 4, 2, and 1), however, Table 3.4 reports the accuracy for four events, a reasonable quantity for concurrent monitoring in most modern processors, even in embedded scenarios [117]. The top accuracy was presented by JRIP (rule-based), followed by four classifiers with the same top-two accuracy: J48 (decision-tree), OneR and PART (rule-based), and Stochastic Gradient Descent (SGD). In this case, most classifiers have accuracy above 80%. Another interesting observation is that lowering the hardware events to fewer than four significantly affects the performance of most classifiers.

The performance penalty with less hardware events was also reported by Torres and Liu [149] in Table 3.3. The authors concentrated on data-only attacks and implemented different experiments on different classifiers, distinguishing between using the complete set of 50 features or a smaller set of 6 features. The results show a very high accuracy on the complete set of features and degradation when only a subset is used.

Table 3.2: Reference studies in HMD, including the full list of targets, classifiers tested, and reference systems. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

Year	Ref.	Target	Classifier	Device	Operating System
2013	[91]	Android malware, Linux rootkits	Decision trees, ANN, KNN, Random Forest	Arm Cortex-A9 OMAP4460, Intel Xeon X5550	Android 4.1.1-1 (kernel 3.2), Linux kernel 2.6.32
2014	[95]	Exploitations on Internet Explorer and Adobe PDF Reader	SVM	Intel IvyBridge Core i7	Windows XP
2015	[156]	Ransomware, password stealers, trojan horses, backdoor, worms	Logistic regression (w/o specialization)	Not specified	Windows 7
2015	[162]	Viruses, worms, trojan horses, spyware, adware, and botnets	ANN	Not specified, Altera EP4CE115	Windows 7
2017	[94]	Malware from various categories, sourced from VirusTotal [163] dataset	Logistic, SimpleLogistic, BayesNet, NaiveBayes, J48, PART, JRIP, OneR, MultiLayerPerceptron, SMO, SGD	Intel Haswell Core i5-4590, Xilinx Virtex 7	Ubuntu 14.04 (kernel 4.4)
2017	[152]	Rootkits	SVM, Decision tree, OC-SVM, Naive Bayes	Intel IvyBridge and Broadwell	Windows 7
2018	[92]	Malware from various categories, sourced from VirusTotal [163] dataset	BayesNet, J48, REPTree, JRIP, OneR, MultiLayerPerceptron, SMO, SGD (w/o ensemble learning based on AdaBoostM1, Bagging)	Intel Xeon X5550, Xilinx Virtex 7	Ubuntu 14.04 (kernel 4.4)
2018	[97]	Malware from various categories, sourced from VirusTotal [163] dataset	Decision trees, Naive Bayes, ANN, KNN, Random Forest (w/o ensemble learning AdaBoost)	AMD Bulldozer	Windows 7
2019	[108]	Malware from various categories, sourced from VX Heaven [164] dataset	J48, IBk, SMO	Intel Sandy Bridge, Haswell, and Skylake	Ubuntu 16.04
2019	[93]	Backdoor, rootkits, viruses, trojan horses	J48, JRIP, OneR, MLP (w/o AdaBoostM1)	Intel Xeon X5550, Xilinx Virtex 7	Ubuntu 14.04 (kernel 4.4)
2021	[146]	Worms, rootkits, viruses, trojan horses	REPTree, JRIP, OneR, MLP, SGD	Intel Xeon X5550, Xilinx Virtex 7	Ubuntu 14.04 (kernel 4.4)
2021	[147]	Stealthy backdoor, rootkits, and trojan horses	DNN	Intel Xeon X5550	Ubuntu 14.04 (kernel 4.4)
2022	[149]	Data-only exploitation	TC-SVM, OC-SVM, LZ78	Intel Nehalem Core i7-920	Ubuntu 16.04 (kernel 4.13)
2022	[158]	Stealthy attack on power grid	SVM	OpenPLC controller with Raspberry PI	8-bus power grid in a PowerWorld simulator

Table 3.3: Summary of best-case performance from reference studies in HMD. HPCs column refers to the number of hardware events the classifiers consider. A is Accuracy, P is Precision, S is Specificity, and F1 is the F1-Score. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

Year	Ref.	Target	HPCs	Classifier	Evaluation Metrics					
					A	P	TPR	S	F1	AUC
2013	[91]	Android malware	6	Decision Tree	-	-	-	-	-	0.83
		Linux rootkits	4	KNN	-	-	0.70[1]	-	-	-
2014	[95]	Internet Explorer exploitation	4	SVM	-	-	-	-	-	1.00
		Adobe PDF Reader exploitation	4	SVM	-	-	-	-	-	1.00
2015	[156]	Ransomware	5	Logistic regression	-	-	-	-	-	0.94
		Ransomware	5	Logistic regression (with Specialization)	0.87	-	0.81	0.96	-	-
2015	[162]	Viruses, worms, trojan horses, spyware, adware, and botnets	5	ANN	-	-	1.00[1]	-	-	-
2017	[94]	Malware from various categories, sourced from VirusTotal [163] dataset	4	BayesNet	0.85	-	-	-	-	-
2017	[152]	Rootkits	16	SVM	1.00	1.00	1.00	-	1.00	-
2018	[92]	Malware from various categories, sourced from VirusTotal [163] dataset	4	J48 (with Ensemble Learning)	0.83	-	-	-	-	0.94
2018	[97]	Malware from various categories, sourced from VirusTotal [163] dataset	6	Random Forest	-	0.86	0.83	-	0.85	0.92
2019	[108]	Malware from various categories, sourced from VX Heaven [164] dataset	5	J48	-	0.82	0.82	-	0.82	0.93
2019	[93]	Backdoor	4	OneR	-	-	-	-	0.94	-
		Rookit	4	MLP	-	-	-	-	0.94	-
		Virus	4	J48 and ensemble learning (AdaBoostM1)	-	-	-	-	0.96	-
		Trojan	4	MLP	-	-	-	-	0.99	-
2021	[146]	Trojan	4	JRIP	-	0.93	-	-	-	-
2021	[147]	Stealthy rootkits	4	DNN	0.93	0.95	0.90	-	0.93	0.98
2022	[149]	Data-only exploits	50[2]	Two Classes-SVM	0.99	-	-	-	-	-
		Data-only exploits	6	LZ78	0.84	-	-	-	-	-
2022	[158]	Stealthy attack on power grid	6	SVM	0.94	-	-	-	-	-

[1] Values extracted from ROC curves considering a false positive rate of 10%.
[2] 50 is the whole set of features. This is why the authors also investigated a reduced set (in the following line).

3.3.2 Efficiency

Alongside the performance, HMD approaches aim to improve the efficiency, which is related to the cost of resources to perform the detection. As this information comes from the hardware layer of the system stack, most studies evaluate

Table 3.4: Performance and efficiency of ML classifiers when applied to HMD [94]. Reproduced from Pegoraro Chenet et al. 2024 [33], licensed under CC-BY.

Classifier	**Accuracy (%)**	**Software**	**Hardware**		
		Latency (ms)	**Latency (ns)**	**Power (W)**	**Area [1]**
BayesNet	81.13	0.624	140	0.44	6794
J48	82.07	0.663	60	0.44	1801
JRIP	83.96	0.653	90	0.44	1504
Logistic	79.24	0.844	340	0.63	13041
MLP	81.13	0.870	40	1.03	36252
NaiveBayes	78.30	0.802	10	1.34	58177
OneR	82.07	0.653	220	0.32	1258
PART	82.07	0.642	680	0.44	2131
SGD	82.07	0.652	340	0.44	2556
SimpleLogistic	79.24	0.648	3020	0.45	4721
SMO	73.58	0.652	2330	0.44	2556

[1] The area is presented as a function of total lookup tables, flip-flops and Digital Signal Processing (DSP) blocks.

FPGA-based implementations, focusing on the latency, power consumption, and area of ML classifiers. In other situations, when the classifier is implemented in software, efficiency analysis usually includes only latency.

Similarly, as reported in the Sub-section 3.3.1, the work of Patel et al. [94] made a comprehensive analysis also of the efficiency of HMD approaches. They analyzed eleven classifiers in terms of latency, power consumption, and hardware cost. The scenario is similar to the one for performance evaluation, except that they also performed hardware implementation to enable additional analysis. They used the Xilinx Virtex 7 FPGA, implemented Weka models in C code, and used the Xilinx High-Level Synthesis (HLS) compiler to generate the final bitstream. The intellectual property cores with the classifiers were synthesized in Vivado for power consumption estimation, considering a 100 MHz clock. Power estimation includes both static power and dynamic power consumption of digital logic. The latency was evaluated both in hardware and software implementations. The software implementation was made at the OS kernel level, and latency here includes the time to read the HPC and execute the classifiers. The Intel Turbo Boost technology was disabled, as it might introduce errors in the time measurement, and the CPU governor was operating at a constant frequency of 800 MHz.

Table 3.4, which summarizes the efficiency analysis from Patel et al. [94], shows the considerable difference between the latencies in software and hardware implementations. Software implementations have latencies almost in the order of milliseconds (ranging from 0.624 ms to 0.870 ms), while hardware implementations are in the order of nanoseconds (ranging from 10 ns to 3020 ns). The authors underlined that these latencies in the order of milliseconds displayed by classifiers in software are high for run-time detection since many malware have execution time

in the range of microseconds or less. They also highlighted that in software implementations, the latency for reading the HPCs may increase significantly when monitoring multiple cores or many counters. Concerning the classification algorithms, BayesNet (Bayesian network), PART (rule-based), and SimpleLogistic (logistic regression) showed the lowest latency in the software implementation. Conversely, NaiveBayes (Bayesian network), Multilayer Perceptron (MLP) (ANN), and J48 (decision tree) are the three low latencies in hardware implementation. This paradox demonstrates the uncorrelation between the algorithms' latencies when comparing implementations at the kernel space and hardware.

A few other works in Table 3.2 also presented latency information for detectors implemented in hardware [92, 93, 146]. All of them revealed latency values ranging from a few nanoseconds to a few microseconds (although, in most cases, the values are in the order of nanoseconds). This is in line with the results of Patel et al. [94] and reinforces the idea that detectors implemented in hardware are at least three orders faster than software implementations.

3.4 Machine learning techniques considerations

In recent years, some studies have explored ML methods to improve the performance of HMD approaches. These techniques mainly aim to compensate for the limited application characterization due to fewer HPCs in the current PMUs. The performance improvements shown by these methods often introduce increased complexity in classifiers, resulting in reduced efficiency, i.e., higher power consumption and increased area requirements. This section discusses ensemble learning, specialization, adaptive detection, and time series ML techniques in HMD. Figure 3.6 presents a visual overview of these techniques.

In **ensemble learning**, multiple ML algorithms are trained independently to create a classifier, combining their outputs to improve decision accuracy [166]. Within HMD, individual algorithms in the ensemble classifier detect various threats, composing a scheme that minimizes the number of hardware events for runtime detection [156, 92, 93]. The resulting performance gains come with increased complexity and efficiency overhead [92, 146]. Two well-known methods in ensemble learning are Boosting (AdaBoost) [167] and Bagging (Bootstrap Aggregation) [168].

Sayadi et al. [92] evaluated the efficiency impact of ensemble learning in a malware detector implemented on Xilinx Virtex 7 FPGA. A case comparing a general classifier with 8 HPCs to a Boosted classifier with 4 HPCs illustrates the significant latency increase. When Boosted, the general MLP algorithm increased the latency from 3020 ns to 5910 ns. Other classifiers increased from 10 ns to 700 ns (OneR) and from 90 ns to 670 ns (J48). In terms of hardware cost, the most significant area increases were observed in OneR (from 2.1% to 5.1%), JRIP (from 2.5% to 5.3%), and BayesNet (from 11.5% to 13.6%). The findings reveal

substantial overhead in both latency and hardware costs.

Specialization is another interesting ML technique. Instead of training a single multiclass classifier to recognize several threat categories, multiple classifiers are trained, each specialized in a specific threat. Authors in [156] were the first to explore specialized detectors in HMD. Using a logistic regression-based classifier for each threat, they reduced the false positive rate by more than half compared to a single detector while increasing the detection rate. In the same paper, they also proposed a two-level detector, mixing a first level based on HMD and a second level based on the software detection approach. The hardware detector was based on specialized ensemble techniques. The latency of this scheme was compared with malware detection purely based on software methods. As a result, they reported that average latency reduced to 1/6.6 when the malware fraction is low and latency reduced to 1/3.1 when 20% of the programs are malware.

In 2019, Sayadi et al. [93] employed specialization to propose a two-stage malware detector, leveraging also ensemble learning techniques. The first stage classifies applications into benign or malware categories (virus, rootkit, backdoor, and trojan horse). The second stage deploys a classifier that works best for each category of malware. Besides the significant improvement in accuracy, they also reported the efficiency overhead of their detector, implemented on Xilinx Virtex 7 FPGA. A comparison of a general classifier with 4 HPCs to a Boosted classifier with 4 HPCs revealed notable latency increases for MLP (from 1.020 to 5.910 μs), OneR (from 10 to 700 ns), J48 (from 30 to 670 ns), and JRIP (from 20 to 560 ns). MLP (from 43.2% to 61.7%), JRIP (from 0.26% to 5.3%), OneR (from 0.49% to 5.1%) and J48 (from 0.93% to 4.3%) exhibited considerable increases regarding hardware cost. The findings highlight the substantial latency and hardware cost overhead due to the specialization and ensemble learning techniques. Moreover, the study of Sayadi et al. from 2021 [147] advanced the specialization technique towards an accurate and run-time stealthy malware detector.

Gao et al. [146] proposed the **Adaptive detection** to optimize the performance versus efficiency. It seeks higher or similar performance as ensemble learning, with a reduced cost. The technique is based on the idea that the algorithm employed in the detector strongly correlates with the nature of the examined malware and the overall performance metric. Adaptive detection involves a dynamic framework that evaluates all underlying ML algorithms in real-time, selecting the optimal classifier to recognize malicious patterns effectively. The implementation encompasses two major online steps: (i) algorithm selection and (ii) malware detection. Consequently, only the most efficient detector is used to differentiate malware from benign applications, eliminating the need to acquire results from individual base detectors and enhancing overall efficiency. The algorithm selection step is done by a lightweight tree-based algorithm that accurately selects the most efficient model for inference. As a result, the scheme showed up to a 94% detection rate while improving the cost-efficiency by more than 5X compared to ensemble classifiers.

The technique based on **time series** employs ML algorithms whose prediction changes if the feature order is scrambled. The classifier does not treat each time point as a separate feature and does not ignore the information contained in the time order of the data. The inspiration comes from nature and human activities, which produce time series, like weather readings, financial recordings, physiological signals, and industrial observations. There are multiple approaches for building time series classifiers: distance-based, shapelet-based, ensembled-based, dictionary-based, interval-based, and deep-learning-based. Particular focus has been given to deep neural networks, especially Fully Convolutional Neural Networks (FCNs) and Long Short-Term Memorys (LSTMs) [169, 170], and scalable time series classification approaches [171, 172].

Time series are particularly interesting to HMD approaches once the program's phase behavior underlies the hardware-based approaches, which are a time series classification problem (see Sub-section 2.3.3). In this direction, Sayadi et al., in [173, 147], proposed a time series approach for detecting at run-time stealthy malware. They employed a FCNs classifier and used just branch instructions as low-level features. Their results indicated the detection of stealthy malware with an average of 94% performance (with only one feature), outperforming the state-of-the-art detection performance. This performance gain comes at a higher computational cost as a deep-learning-based solution.

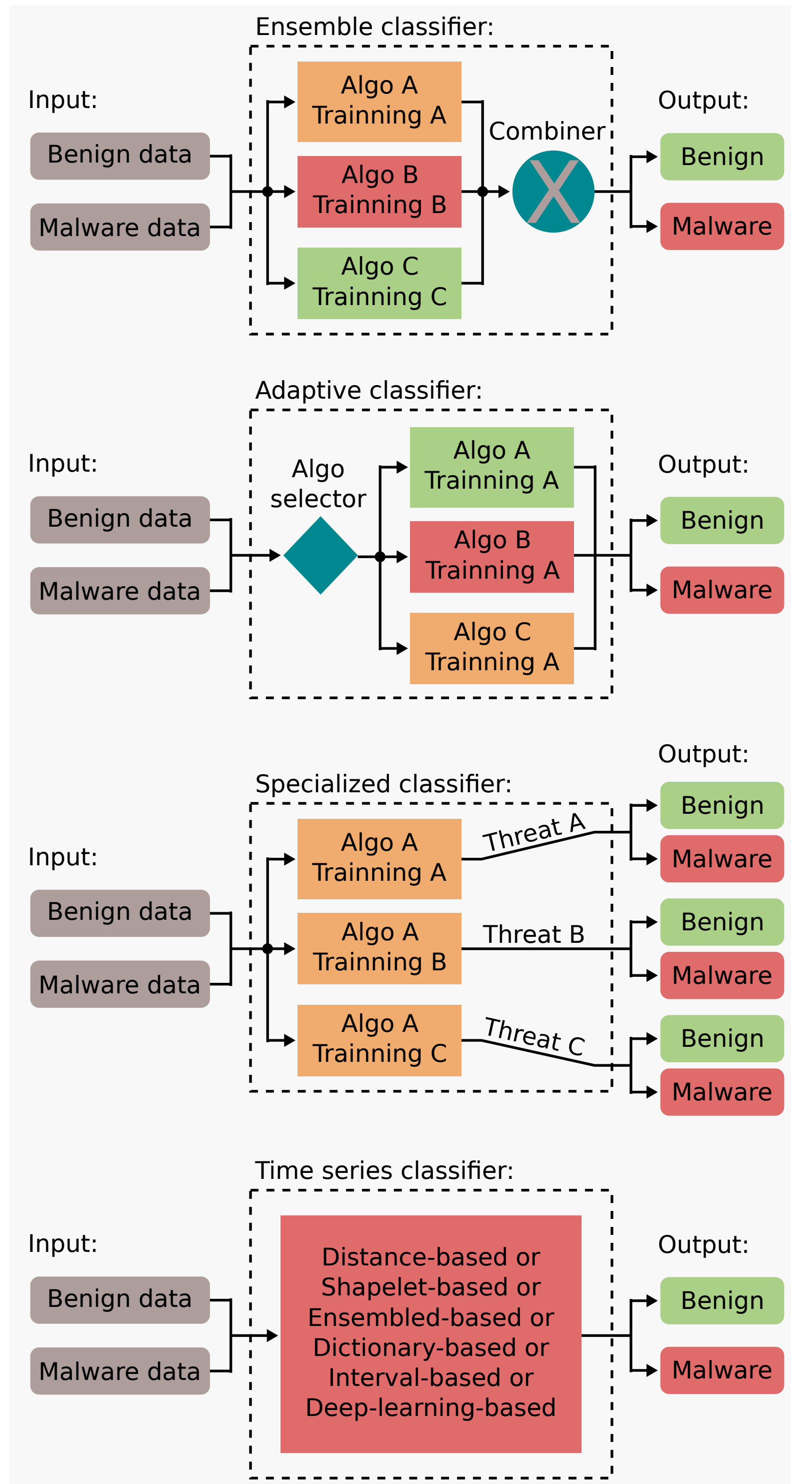


Figure 3.6: Machine learning techniques employed to improve the performance of HMD approaches.

Chapter 4

Zero-day hardware-based detection of stack buffer overflow attacks

©2025 Institute of Electrical and Electronics Engineers (IEEE).
Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo, "Zero-Day Hardware-Supported Malware Detection of Stack Buffer Overflow Attacks: an application exploiting the CV32e40p RISC-V core," in IEEE 26th Latin American Test Symposium (LATS), 2025, doi: 10.1109/LATS65346.2025.10963939.

SBO [63] is one of the techniques that enable the common control-flow attacks (see Sub-section 2.2.2), employing buffer overflows, i.e., when a program writes more data to a buffer than it can hold, overwriting the adjacent memory space. The canonical SBO attack is characterized by nonvalidated data overwritten by the function return address in the memory stack, as illustrated in Figure 4.1. Properly setting the function return address enables redirecting the program's execution flow to a malicious code. This chapter presents a case study which explores the capability of hardware-based approaches to support zero-day detection of SBO attacks. The target platform employed was **RISC-V** in an attempt to disseminate the HMD to the community surrounding this platform, which still does not fully explore this domain, as can be noted in Table 3.2.

Section 4.1 expands the discussion of anomaly detection as a ML technique. Section 4.2 presents the methodology. Section 4.3 discusses the experimental results. And Section 4.4 briefly explores autoencoders to improve the detector performance.

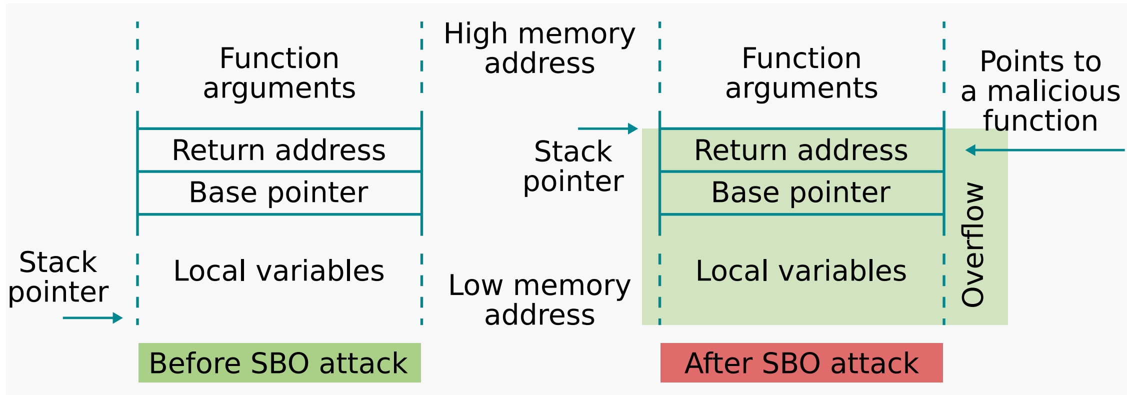


Figure 4.1: Canonical SBO attack flow. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

4.1 Anomaly detection enabling zero-day malware detection

In the field of ML, **anomaly detection** involves identifying patterns in data that deviate from expected behavior. This technique is important because anomalies in data often represent significant and frequently critical, actionable information. It is employed in various applications, such as fraud detection for credit cards, insurance, or health care, fault detection in safety-critical systems, military surveillance for enemy activities, and intrusion detection for security [12]. Anomaly detection techniques can operate in supervised, unsupervised, and semi-supervised learning modes. In supervised, the training dataset has labeled instances for normal and anomaly classes. In unsupervised, also known as outlier detection, the samples in the training dataset do not have labels. In semi-supervised learning mode, also known as novelty detection, the training dataset has only samples labeled with a normal class. In this chapter's case study, novelty detection is employed (Figure 4.2). The classifier is trained only with benign applications (white dots) to learn a frontier (red circles). New observations inside the red circles (purple dots) are classified as benign applications, and outsiders (yellow dots) are malicious applications. This detection behavior enables an exciting advantage, which is zero-day malware detection.

In HMD, some studies employed the anomaly detection technique, such as the one summarized in Table 4.1. They reported a wide performance range, from 0.75 (accuracy) to 1 AUC. Regarding ML detection, most focus on a single classifier, lacking a comprehensive investigation of different algorithms. **One-class Support Vector Machines (OC-SVM)** [176, 177] is well known for anomaly detection. As a one-class classifier, it is trained with only one class of data to distinguish whether or not a new observation belongs to that class. OC-SVM is a member of the family

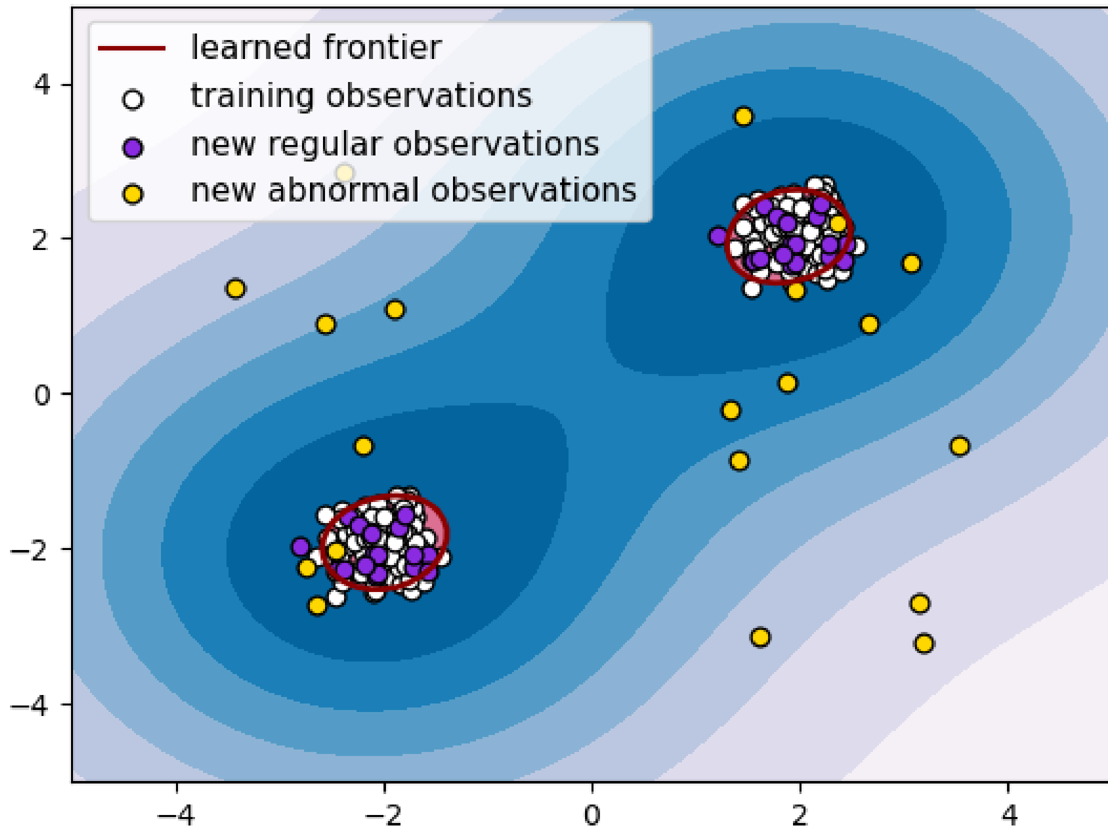


Figure 4.2: Anomaly novelty detection [175].

of SVM classifiers. These statistical approaches target to find a decision boundary (hyperplane) that best separates data points belonging to different classes. Hyperplane, support vectors, and kernel trick explain the algorithm. The hyperplane is the decision boundary that separates different classes. For example, for a binary classification, the hyperplane is just a line, in three dimensions the hyperplane is a plane, and in higher dimensions a hyperplane is used. The support vectors are data points closest to the hyperplane for defining the best decision boundary. The kernel trick allows SVM to handle linear and non-linear decision boundaries. **Local Outlier Factor (LOF)** [178] is also well known for anomaly detection, sometimes distinguishing data points better than OC-SVM [179]. It tries to find anomalous data objects based on the concept of local density, where locality is given by k nearest neighbors. The distance to those neighbors is taken into account to estimate the density. By comparing an object's local density with its neighbors' local densities, regions of similar density can be identified. Objects with a lower density than neighbors can also be detected and considered outliers/anomalies. **Lempel-Ziv Data Compression (LZ78)** [180] is, actually, the principle of compression schemes like GIF, PNG, and ZIP. As a classifier, input data is transformed into

symbols, the LZ78 is applied to obtain a tree, and probability values are assigned to each node, which allows pattern detection.

Table 4.1: Reference studies in anomaly HMD. HPCs column refers to the number of hardware events the classifiers consider. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

Year	Ref.	Target malware/exploit	HPCs	Classifier	Accuracy	AUC
2014	[95]	ROP on Internet Explorer	4	OC-SVM	-	1.00
		ROP on Adobe PDF Reader	4	OC-SVM	-	1.00
2015	[179]	ROP and SBO on web browser	6	LOF	-	-
2017	[152]	Rootkits	16	OC-SVM	0.75	-
2022	[149]	Data-only exploits	50	OC-SVM	0.91	-
		Data-only exploits	6	OC-SVM	0.75	-
		Data-only exploits	50	LZ78	0.92	-
		Data-only exploits	6	LZ78	0.84	-
2022	[158]	Stealthy attacks on power grid	6	OC-SVM	0.94	-

Isolation Forest and Elliptic Envelope are other classifiers for anomaly detection, although not yet explored in HMD approaches. **Isolation Forest** [181] is based on the fact that anomalies are data points that are few and different, applying binary decision trees over this idea. Samples are isolated by randomly selecting a feature and then selecting a split value between the maximum and minimum values of the selected feature. Since a tree structure can represent recursive partitioning, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of such random trees, measures normality. A shorter path length for a particular sample means it is likely to be an anomaly. **Elliptic Envelope** [182] creates an elliptical boundary around the data, considering anything outside this envelope as an anomaly.

4.2 Methodology

A specific pipeline is proposed to evaluate the effectiveness of HMD, as shown in Figure 4.3. A simulation environmental (red box) uses a microarchitectural simulator (`GVSoC` [144]) to run the applications on the CV32E40p RISC-V core and collect the HPCs. A coupled analysis framework (green box) implements the HMD, from feature selection to classification, targeting the attack detection.

In terms of analysis techniques that enable attack detection, several studies sample the HPCs multiple times throughout the application's execution, like in [95, 179, 149, 158], from Table 4.1. We proposed to sample only at the end of the application's execution. ML algorithms are trained on executions with random inputs, enabling attack detection regardless of the application input. While this method makes detecting subtle changes in software behavior more challenging, it

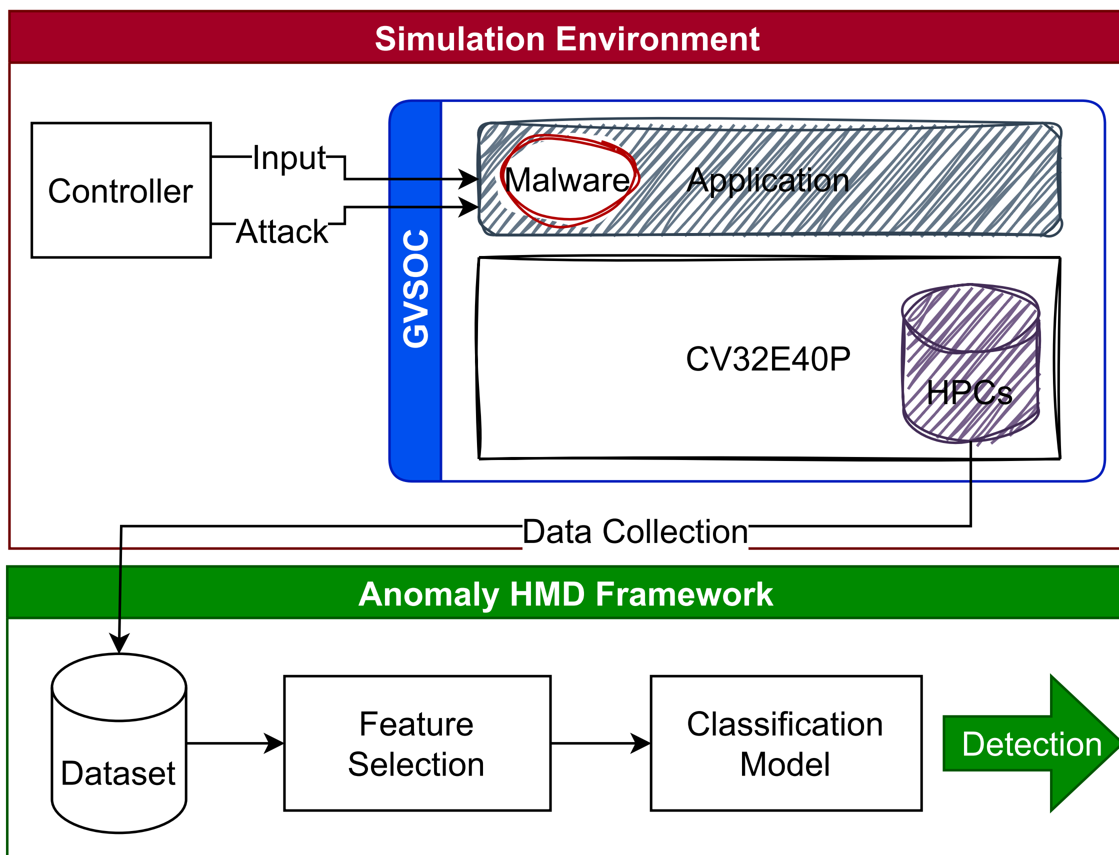


Figure 4.3: Methodology overview for the zero-day detection of SBO attacks. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

significantly reduces the number of ML inferences, thereby minimizing performance overhead on the host computer. Additionally, this method is particularly suited for situations with a defined endpoint. In other cases, methods based on time or event domains may be more appropriate [116].

4.2.1 Simulation environment

The simulation environment comprises the controller scripts and the microarchitectural simulator (`GVSoC`) where the applications run. The controller scripts manage the experiments by (i) mutating the application under attack to include the malware, (ii) injecting random input stimulus in the application, (iii) compiling it to run on the simulator, and (iv) running the simulation. The mutation is done by inserting the malware function and selecting where the SBO must be triggered. To evaluate different attack scenarios, three execution modes are implemented (as illustrated in Figure 4.4): (i) **Benign**, where no malware is present; (ii) **Overt**, where, upon a successful attack, control transfers to the malware and never returns

to the original caller; and (iii) **Stealthy**, where a final jump back to the caller allows the malware to conceal its presence.

In an Overt mode closer to a real scenario, it may generate doubt about when to trigger the HPCs collection since the application never reaches the end. However, there are multiple techniques to detect the interruption of an application. For example, in bare-metal, a watchdog may be employed; in an OS environmental, the process status may be analyzed. In an overt mode closer to a real-world scenario, determining when to trigger HPCs collection may generate doubt, as the application never reaches the end. However, multiple techniques can detect application interruptions. For instance, in a bare-metal environment, a watchdog timer can be used, while in an OS environment, monitoring the process status can provide the necessary insight.

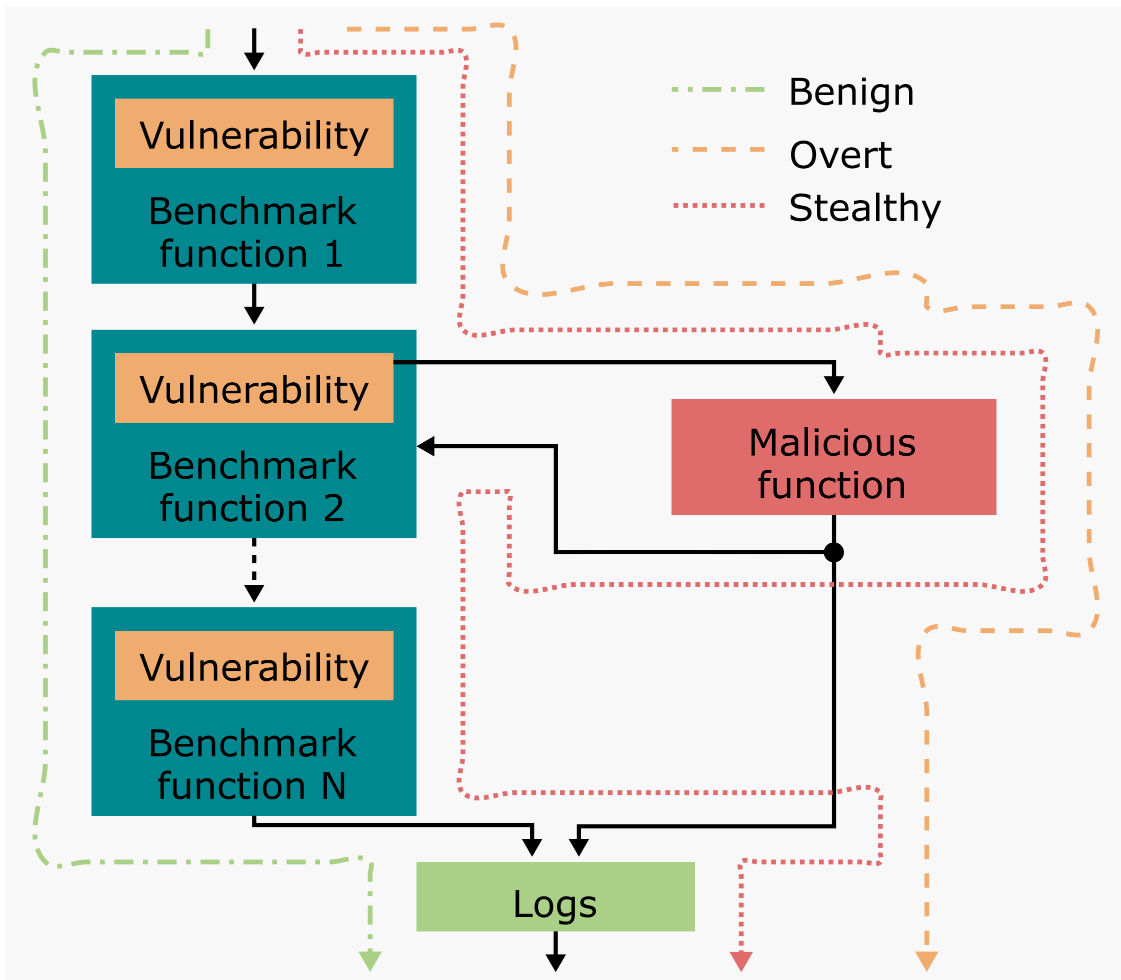


Figure 4.4: Execution modes configuring different attack scenarios. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

In the applications running on the simulator, SBO vulnerabilities are artificially introduced to emulate potential vulnerabilities that programs may have. They consist of a `vuln` function shown in the Listing 4.1, inserted arbitrarily before the call of some functions that compose the application. When entering in the `vuln` function, the input word is copied to the buffer, simulating regular data input by the user. Later, variables `mode`, `inj_place`, and `chunk_inj`, configured by the controller scripts define if the attack is triggered. If positive, the buffer overflow writes the pointer to the malicious function in positions 76 to 79 of the buffer. As the buffer has only 64 positions, the positions 76 to 79 are the function return address in the memory stack. Upon returning from `vuln`, the program's execution flow will be redirected to the malicious function.

Listing 4.1: Vulnerability function which enables the SBO attack. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

```
void vuln(char *input, int ip){

    //copy the input to the buffer
    char buffer[64];
    memcpy(buffer, input, 64);

    //save return address
    if (mode == 's') {
        return_addr[0] = buffer[76];
        return_addr[1] = buffer[77];
        return_addr[2] = buffer[78];
        return_addr[3] = buffer[79];}

    //inject buffer overflow
    if ((mode == 'o' || mode == 's') &&
        (orig_place == inj_place) &&
        (current_chunk == chunk_inj)) {
        sf_addr = (uint32_t) &malicious_function;
        buffer[76] = sf_addr & 0xFF;
        buffer[77] = (sf_addr >> 8) & 0xFF;
        buffer[78] = (sf_addr >> 16) & 0xFF;
        buffer[79] = (sf_addr >> 24) & 0xFF;}
}
```

4.2.2 Hardware-based malware detection framework

The HMD framework consists of feature extraction, feature selection, classification model training, and finally, enabling the performance analysis. The **feature extraction** in this case study used the Application Programming Interface (API) provided by the simulator to collect the HPCs at the end of the application's execution. As the output of this step, three different datasets are generated: (i) benign data used for training, (ii) stealthy executions, and (iii) overt executions. The last two are used for testing and comprise benign and malignant data.

The **feature selection** was performed in two consecutive steps: (i) a manual pruning on the complete list of extracted HPCs; and (ii) a feature ranking elaboration based on the PCA technique. The manual pruning is due to prior knowledge

from the literature that certain HPCs have limited value in providing insights into program behavior [90, 94]. For example, counters that track the total number of cycles (`CYCLES` HPC) or count active cycles (`ACTIVE_CYCLES` HPC) do little more than track time, providing minimal insight into the performance and efficiency of the analyzed code. Therefore, `CYCLES` and `ACTIVE_CYCLES` were manually removed, together inactive (zero-locked) HPCs.

In the second step in feature selection, the PCA technique [176, 177] is employed to rank the remaining HPCs. It is a widely used unsupervised machine learning method consisting of the linear decomposition of original features as linear combinations of a set of basis vectors, which are the principal components. The principal components are composed of the eigenvectors and eigenvalues. Eigenvectors represent the directions or axes of maximum variance in the data, while eigenvalues represent the magnitude of variance along those directions. PCA can make rankings based on the eigenvalues associated with the features and the principal components. The manual pruning and ranking of HPCs result in a streamlined set that retains significative counters ordered by priority for detecting malicious programs, significantly reducing the complexity of the analysis.

As **classification models**, this case study analyzes the following four algorithms for anomaly detection (detailed in Sub-section 4.1): OC-SVM, LOF, Isolation Forest and Elliptic Envelope.

4.3 Results

Figure 4.5 shows the setup for simulations of zero-day detection of SBO attacks. All experiments run on a **host computer** with Ubuntu 20.04.6 LTS (Focal Fossa) in different hardware configurations: AMD Ryzen 7 5700U and AMD Ryzen 9 7950X (for fast simulations). On this computer, the **`GVSoC`** [144] is employed, an event-driven simulator with a hardware-oriented description. It simulates the behavior of the **CV32E40P** (also called RI5CY), a 32-bit CPU core with four pipeline stages that implement the RV32IM[F|Zfinx]C RISC-V specification and Parallel Ultra Low Power (PULP) custom extensions [183]. This CPU supports tracking of seventeen HPCs, and the `GVSoC` simulator models all of them.

In this environment, the target applications are executed. They come from the MiBench suite [165] and are C code compiled via the PULP Toolchain to run in bare-metal in the simulator. The set of applications comprises **Advanced Encryption Standard (AES) encryption**, **Rivest–Shamir–Adleman (RSA) encryption**, **Secure Hash Algorithm (SHA) hash**, and the **Dijkstra algorithm**. The choice for these applications was guided by the limitations imposed by the simulator in dealing with more complex applications. `GVSoC` has restrictive limitations in supporting the C standard library and emulating an I/O complete system. Each application was manually annotated to artificially include the

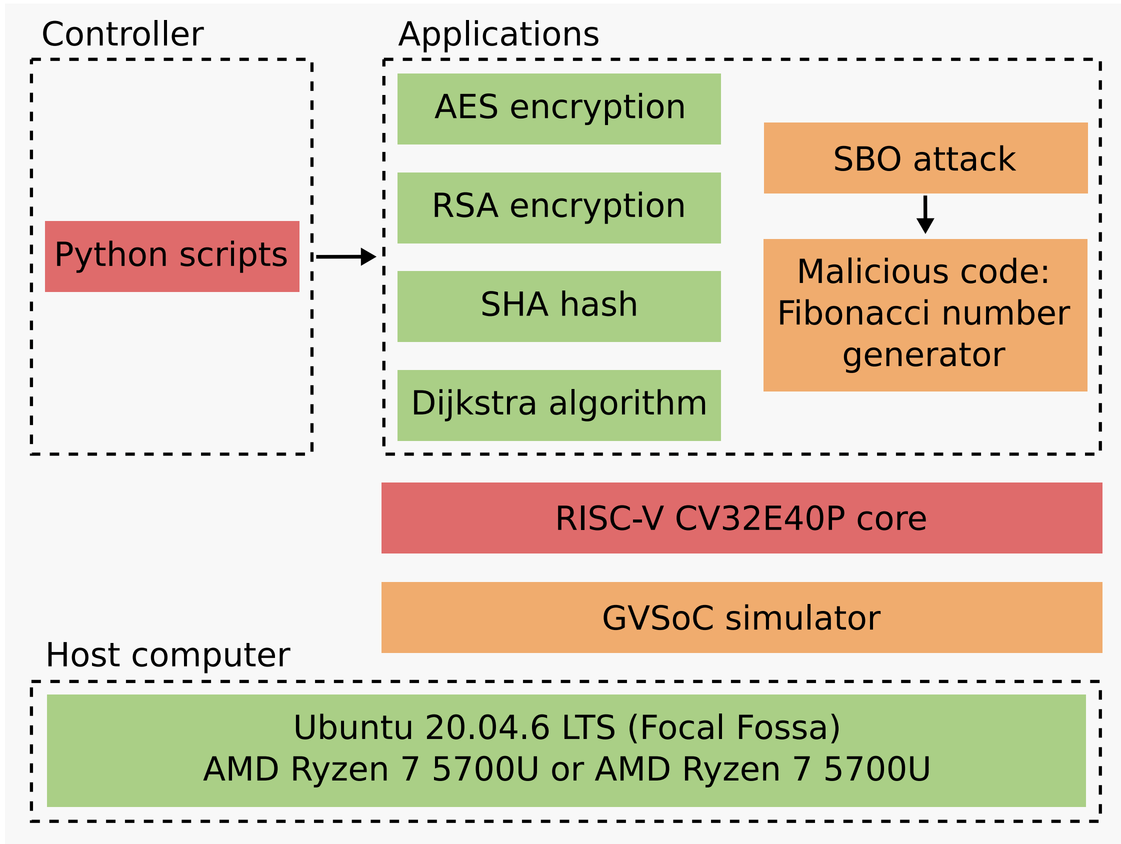


Figure 4.5: Setup for simulations of zero-day detection of SBO attacks.

SBO attack and include the code snippet to emulate the malicious function. As the specific malware operation is not the goal of the detection, but the detection performance depends on malware execution, the Fibonacci number generator was employed to emulate the malicious program. The dependence of the malware for the detection is explained by the slight deviation in the HPCs if a smaller number of extra instructions is introduced, making the SBO attack detection challenging.

The **Fibonacci algorithm** generates N Fibonacci numbers. It allows the creation of multiple versions of the same code with varying sizes by simply adjusting the sequence length. Thus, the controller scripts can impose different code size ratios between the application and the malware by selecting the Fibonacci sequence length. The metric employed is the number of executed instructions by the code, assessed through the `INSTR` HPC. The following results are shown considering two different code size ratios: malicious code with 1% and 10% of the application. To implement these ratios, the estimation of the applications is implemented as a function of the input (in bytes) for AES and SHA and as a function of the number of nodes for Dijkstra. For RSA, estimating the application size is more complex due

to its dependency on randomly generated prime numbers. Therefore, an initial execution is required to measure the application size, followed by a second execution for data extraction.

The **controller** is written in Python and runs directly on the host computer. With the setup working, simulations are performed. For each application and attack execution mode (Stealthy and Overt), we conducted 20K executions (10K for training and 10K for testing), randomly varying the application input stimulus. During testing, 50% of executions were in Benign mode, and 50% were with attacks. In the HMD framework, the feature selection (PCA technique) and the classification models are implemented using the `Scikit-learn` library for Python. As a performance evaluation metric, only accuracy is shown for simplification. But precision, recall, specificity, F1-score, and AUC were also evaluated and did not reveal any novelty related to the accuracy.

4.3.1 Feature selection

Table 4.2 presents the result of feature selection: the HPCs remained after manual pruning and ranked according to PCA. The ranking for a given target application is the same, independent of the execution mode (Overt or Stealthy). It is worth noticing that the `INSTR`, `RVC`, and `LD` HPCs are, respectively, the most significant HPCs for all the applications. The following sub-sections use this ranking to present the detection performance according to an incremental number of HPCs, ensuring future benchmarks even in situations with restrictions in the number of HPCs.

Table 4.2: Result of feature selection, with HPCs remained after manual pruning and ranked according to PCA. Index 1 defines the most significant counter. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

HPC	Description	AES	RSA	RSA[1]	SHA	Dijkstra
`INSTR`	Number of instructions executed	1	1	1	1	1
`LD_STALL`	Number of load data hazards	4	7	6	4	5
`LD`	Number of data memory loads executed	3	3	3	3	3
`ST`	Number of data memory stores executed	5	5	5	5	6
`JUMP`	Number of unconditional jumps (j, jal, jr, jalr)	6	8	8	6	8
`BRANCH`	Number of branches (taken and not taken)	7	4	4	7	4
`BTAKEN`	Number of taken branches	8	6	7	8	7
`RVC`	Number of compressed instructions executed	2	2	2	2	2

[1] Refers to RSA implementation with constant prime numbers.

4.3.2 Performance in stealthy mode

Figure 4.6 shows the detection accuracy for Stealthy mode. For each classifier and application, four bars refer to accuracies related to 1, 2, 4, and 8 HPCs (from

left to right in the figure, respectively). When using a single HPC, it is employed the counter with index 1 in Table 4.2 (i.e., `INSTR`); when using two, the HPCs with indexes 1 and 2 (i.e., `INSTR` and `RVC`), and so on. Analyzing the overall performance in AES, SHA, and Dijkstra, the accuracy is higher than 90% even when the size of the malicious function is 1% of the application. Despite this good performance, attacks in RSA are only detected when the ratio reaches 10%. We deeply investigated this application and found that the random search for prime numbers in generating public and private keys results in massive variability in the HPCs, impacting the detection performance. To confirm the hypothesis, we modified the application, forcing the prime number to a constant value, and repeated the simulations. The results are in Figure 4.7, showing a drastic performance increase (accuracy close to 100%) and also demonstrating that randomness in the application algorithm might impact HMD.

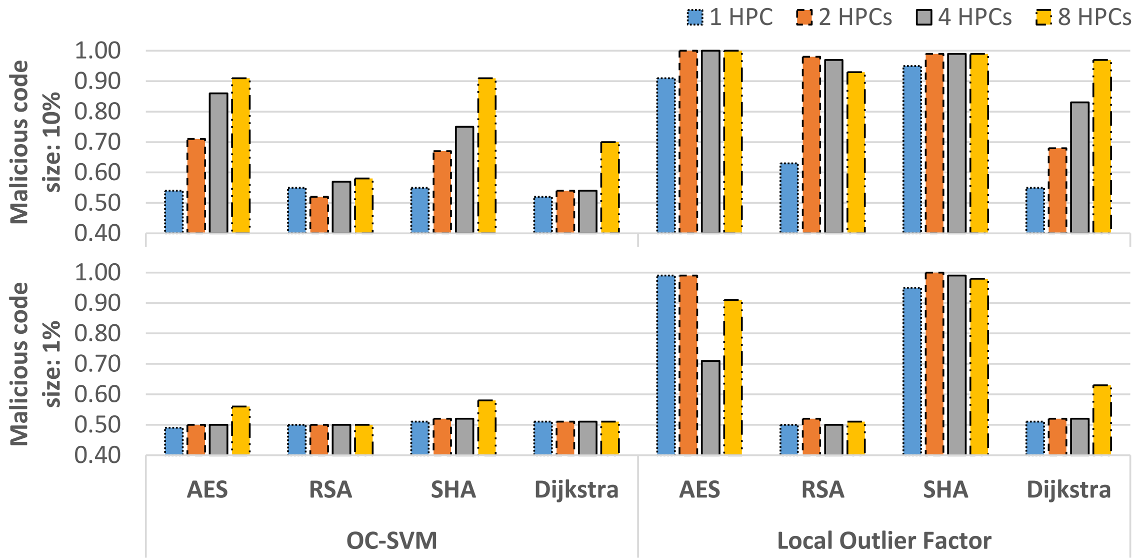
Figure 4.8 visually investigates the detection performances on the RSA application using the PCA decomposition in the three first principal components when malicious function size is 1%. The clear separation between normal data (green dots) and attack data (red dots) in RSA with constant prime numbers, compared to the partial overlap in the original application, explains the improved performance.

Turning attention to the classifiers, it is clear that the LOF classifier stands out as a powerful tool. Unlike the others analyzed, it consistently achieves accuracy close to or exceeding 90% when a small number of HPCs is considered. For example, it performs well with AES, RSA with constant prime numbers, and SHA with just one HPC and a malicious function size of 1%. Moreover, the Elliptic Envelope also achieves accuracies higher than 90% in more challenging cases requiring multiple HPCs. Conversely, the Isolation Forest could not detect the attacks. This weak performance is likely due to the masking problem, where numerous anomalies hide their presence, making it difficult for the classifier to effectively isolate individual anomalies [181, 184].

Ultimately, we evaluated the capacity of training a single classifier to detect attacks among all applications. However, the final detection performance was poor. The only case with a significant performance was using LOF to classify a 10% malicious function size: 88% accuracy with two HPCs and 90% accuracy with four and eight HPCs. This performance suggests that using specialized detectors for each application is the most effective solution.

4.3.3 Performance in overt mode

In this execution mode, the results are also expressed based on the malicious function size. Since the controller scripts randomly determine the attack's launch point, and the program terminates immediately after executing it, the application may not fully process its input data. As a result, the ratio between the size of the



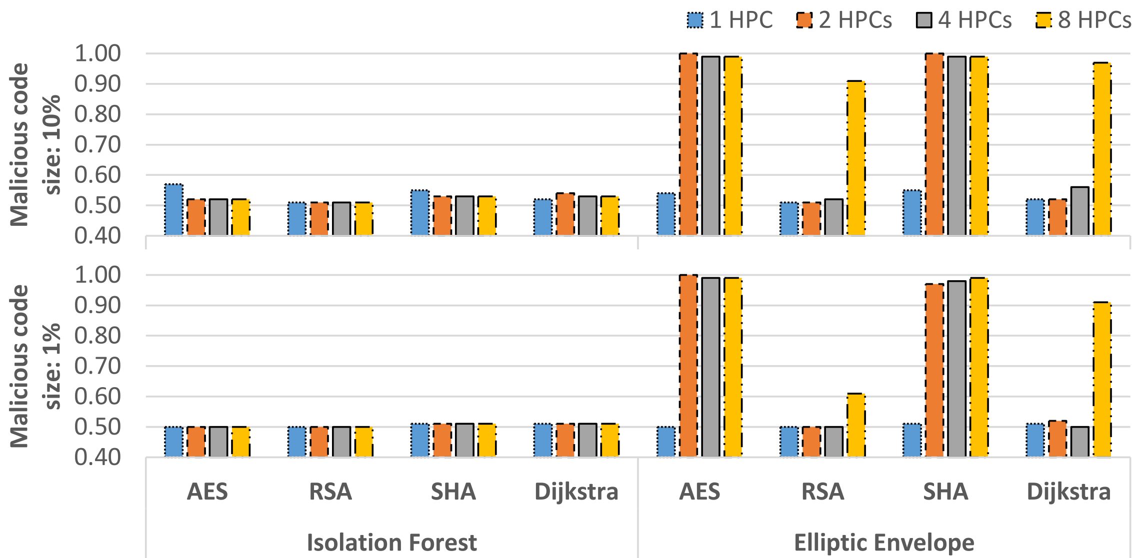


Figure 4.6: Accuracy for Stealthy mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

malicious function and the application may, eventually, exceed the nominal stated value.

Figures 4.9 and 4.10 show the detection accuracy for Overt mode. In AES, RSA with constant prime numbers, SHA, and Dijkstra, the accuracy is at least 94% when the malicious function size is 1% of the application. Similarly to Stealthy

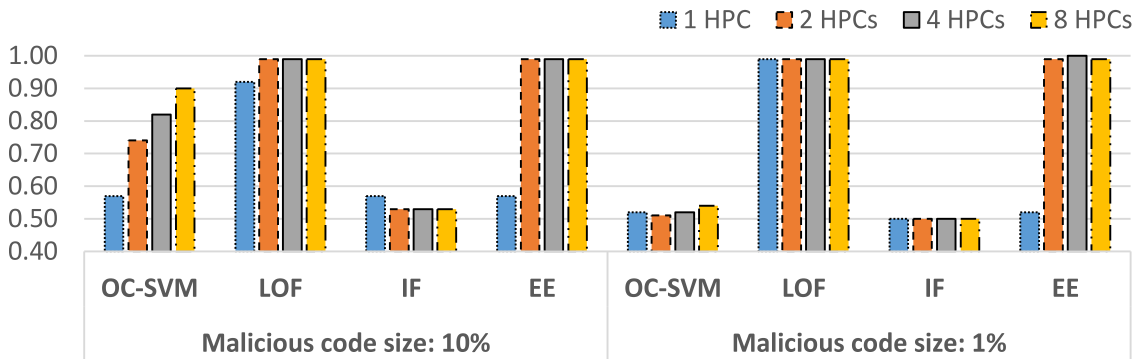


Figure 4.7: Accuracy for RSA with constant prime number, in Stealthy mode. IF is an Isolation Forest, and EE is an Elliptic Envelope. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

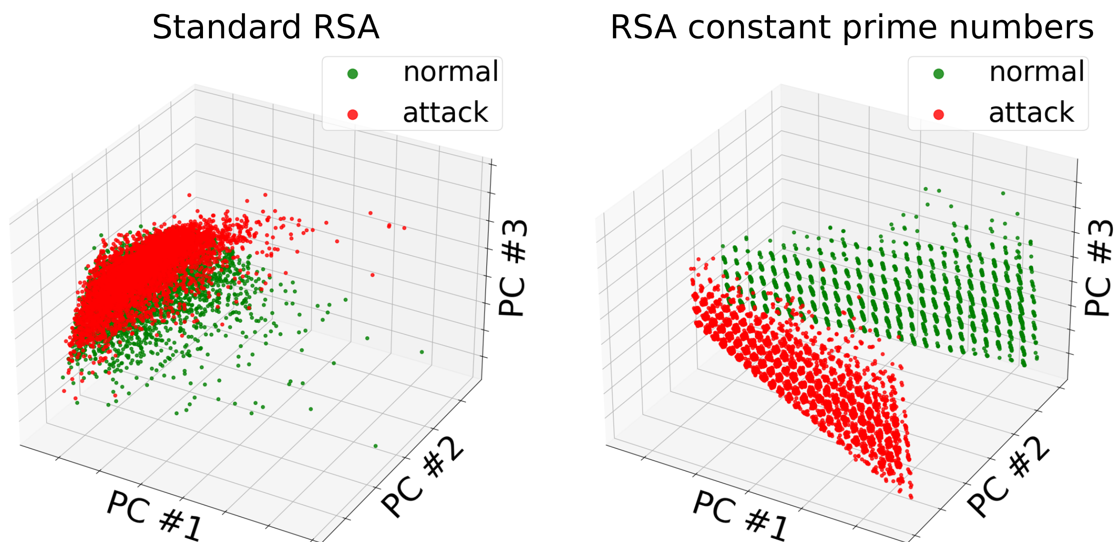
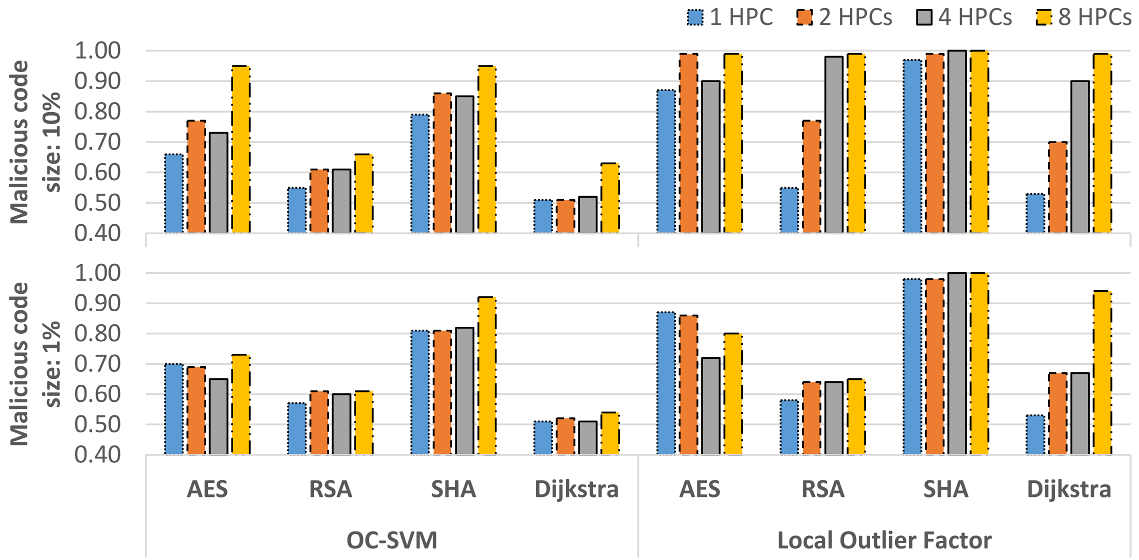


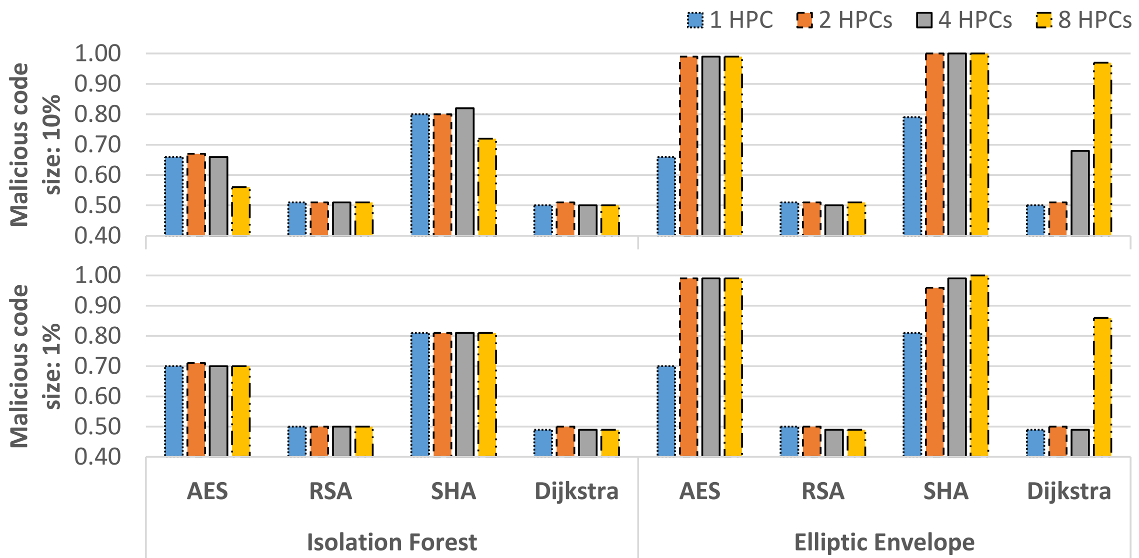
Figure 4.8: RSA related PCA decomposition in the three first principal components, in Stealthy mode. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

mode, the accuracy in the original RSA implementation is poor. The accuracy is 65% with a malicious function size of 1%, improving to 99% just when the size is 10% and the number of HPCs is eight. The challenge in this detection is once more visible through the PCA decomposition of data (Figure 4.11), which shows the overlapping of normal data (green dots) and attack data (red dots), being an obstacle to detection. As discussed in Sub-section 4.3.2, the responsible for this behavior is the random search for prime numbers in the generation of public and private keys, which increases the variability in the HPCs.

Compared to Stealthy mode, the detection performance is slightly better in



(a)



(b)

Figure 4.9: Accuracy for Overt mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

Overt mode. For example, considering the malicious function size of 1%, there are cases where the accuracy jumped from close to 50% (in Stealthy mode) to close to 70-80 and 90% (in Overt mode): AES (OC-SVM and Isolation Forest) reached close to 70%, SHA (Isolation Forest) reached close to 80%, and SHA (OC-SVM) and Dijkstra (LOF) reached close to 90% of accuracy. On the other hand, there are

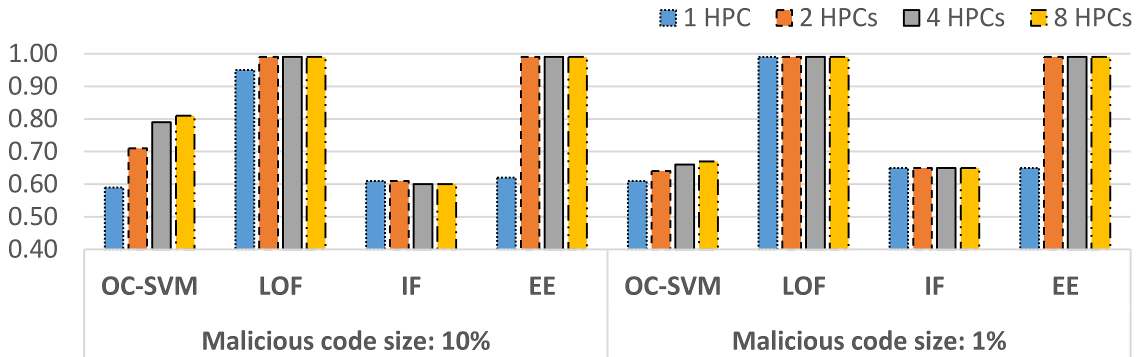


Figure 4.10: Accuracy for RSA with constant prime number, in Overt mode. IF is an Isolation Forest, and EE is an Elliptic Envelope. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

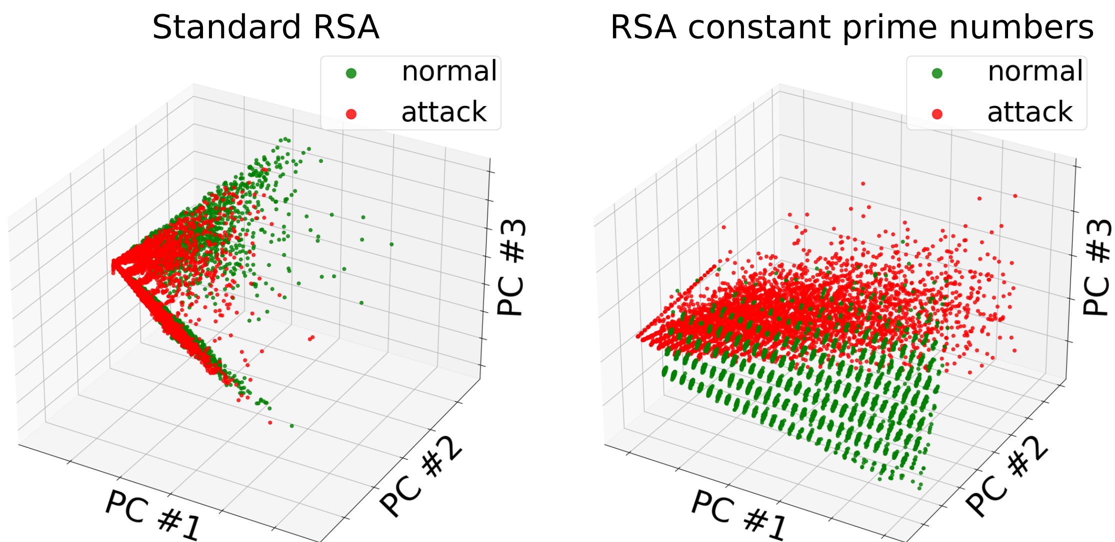


Figure 4.11: RSA related PCA decomposition in the three first principal components, in Overt mode. Reproduced from Pegoraro Chenet et al. 2025 [174], licensed under CC-BY.

also a few cases where the performance in Overt is lower than in Stealthy mode, like in the AES application (LOF classifier). The overall better performance in Overt mode is attributed to the possible cases in which the execution of the application is interrupted, significantly impacting the HPCs and improving the detection.

When comparing classification algorithms, LOF and Elliptic Envelope performed the best again. Eventually, we also evaluated the capacity in the Overt mode of training a single classifier to detect attacks among all applications. Again, the results suggested the necessity of specialized detectors for effective detection: the best accuracy for malicious function size of 10% is 83%, and with malicious

function size of 1% is 78%, both with LOF classifier.

4.4 Brief exploration of autoencoders

Results in the previews 4.3.3 and 4.3.2 Sub-sections revealed the infeasibility of anomaly HMD to detect the attack in the original RSA application when the malicious function size is 1%. This section explores autoencoders to improve the performance of the hardware-based detector, reversing the result toward a feasible detection in the original RSA.

The autoencoder is a ML method used as a preprocessing stage before classification and learning features to increase accuracy. A penalty in this method relies on the time to train the autoencoder, which is considerably higher than the traditional machine learning classifiers. Inside, the autoencoder is an unsupervised feedforward neural network trained to attempt to copy its input to its output. The network is composed by an encoder $h = f(x)$ and a decoder $r = g(h)$ that performs the input reconstruction $g(f(x)) = x$. The hidden layer on the limit between the encoder and decoder is constrained to have a narrow bottleneck (layer with dimension lower than the input), which forces the autoencoder to learn the most relevant aspects of the data. With just a single layer each, the encoder and decoder can represent an approximation of any function to an arbitrary degree of accuracy. However, deep encoders and decoders offer many advantages. An autoencoder with two hidden layers (besides the bottleneck) can approximate any mapping from input to output arbitrarily well [145].

The addition of an autoencoder in the case study of this chapter modifies the methodology overview shown in Figure 4.3. Figure 4.12 presents the updated anomaly HMD framework, including this new ML method. After the feature selection, the autoencoder is individually trained using the encoder and the decoder. Once the autoencoder is trained, the decoder is discarded, and the encoder output gives data to train the classifier. With the autoencoder and classifier trained, the framework is ready for testing. The testing dataset feeds the encoder, and its output is used in the classifier to perform the detection.

To implement the autoencoder, this study used `Keras`, an open-source library that provides a Python interface for ANNs. Considering N the number of features or HPCs, the autoencoder architecture is presented in Figure 4.13, representing in the drawing the number of neurons for a case with eight features. As a feedforward neural network, information moves only from input to output direction, and each neuron is connected to every neuron in the adjacent layer. In `Keras`, this structure is represented by the `Dense` class. The encoder and decoder have two hidden layers because Goodfellow et al. [145] defended that two hidden layers can approximate any mapping from input to output arbitrarily well. In the encoder, the first hidden layer has $2 * N$ neurons and the second hidden layer N neurons, followed by the

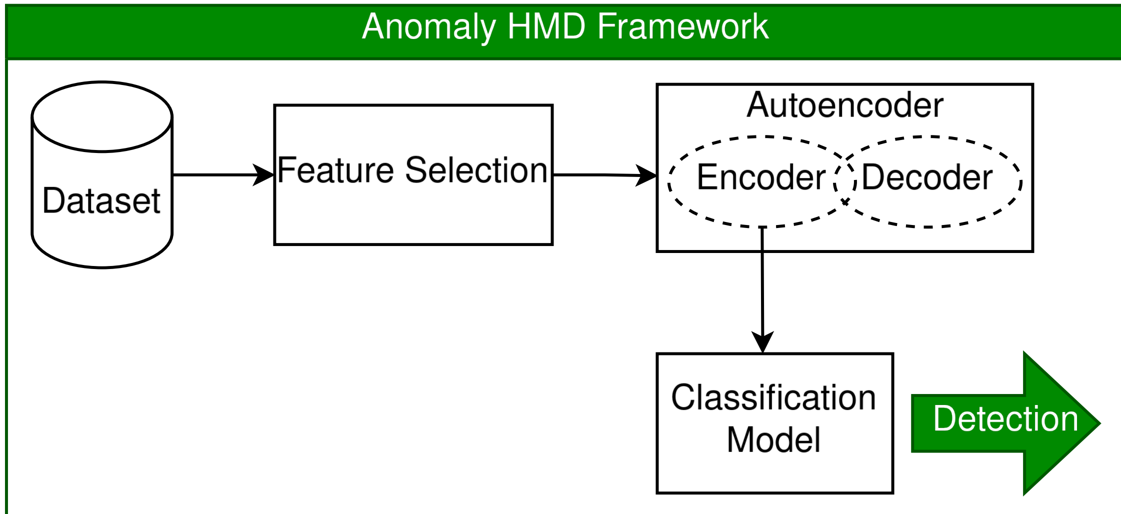


Figure 4.12: Anomaly HMD framework including the autoencoder.

bottleneck with N neurons. In the decoder, the first hidden layer has N neurons and the second hidden layer $2 * N$ neurons. The activation function used in all hidden layers is the LeakyReLU, and in the rest of the layers, no activation is applied ($a(x) = x$). Regarding hyperparameters, the learning rate was set to 0.00001, the optimizer was Adam, and the loss was a mean squared error. The autoencoder was trained with 200 epochs and a batch size of 16. This configuration demands a considerable time to train the autoencoder (compared to the time to train a traditional ML classifier), but we consider this not critical.

Figure 4.14 presents the accuracy in Stealthy mode when the proposed autoencoder configuration is added as a preprocessing step. Straightforwardly, the addition of the autoencoder does not reverse the detection of the attack in the original RSA application, always remaining with accuracy close to 50% when the malicious function size is 1%. The implementation was leveraged to extend the performance analysis to all the applications, not just in this critical case. The comparison with Figure 4.6 shows that overall, there is no significant gain with the addition of the autoencoder. The cases where there is a considerable accuracy increase are punctual, and there are also cases when the accuracy is significantly reduced. It is worth noticing that the OC-SVM classifier only increases its performance with the employment of the autoencoder. For instance, with a malicious function size of 1%, AES with eight HPCs; and with a malicious function size of 10%, SHA and Dijkstra (with four HPCs and eight HPCs, respectively). Conversely, the Elliptic Envelope mostly decreases its performance, mainly notable when the malicious function size is 1% of the application. These results suggest that the performance of each classification algorithm may react differently with the employment of autoencoders.

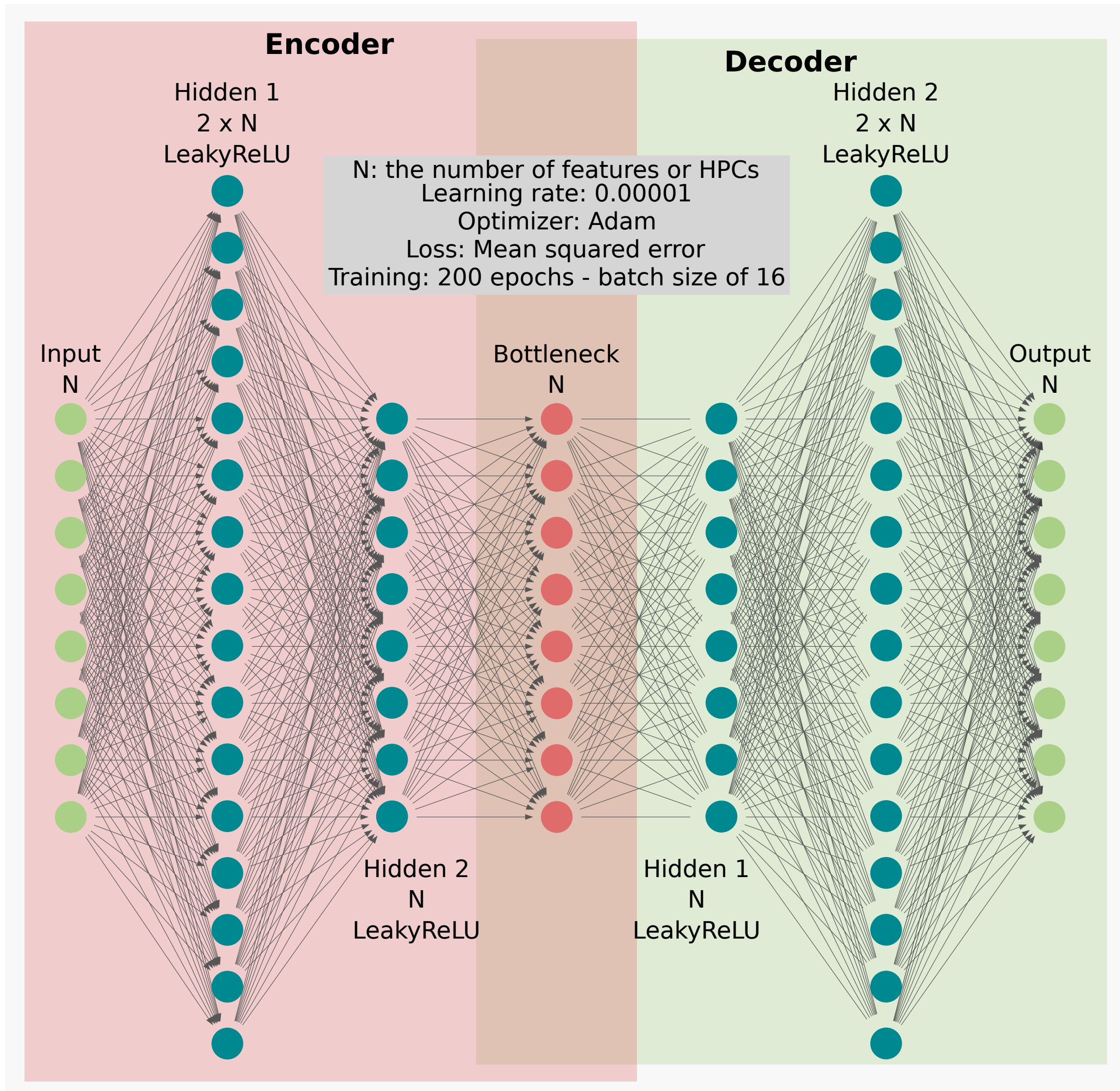
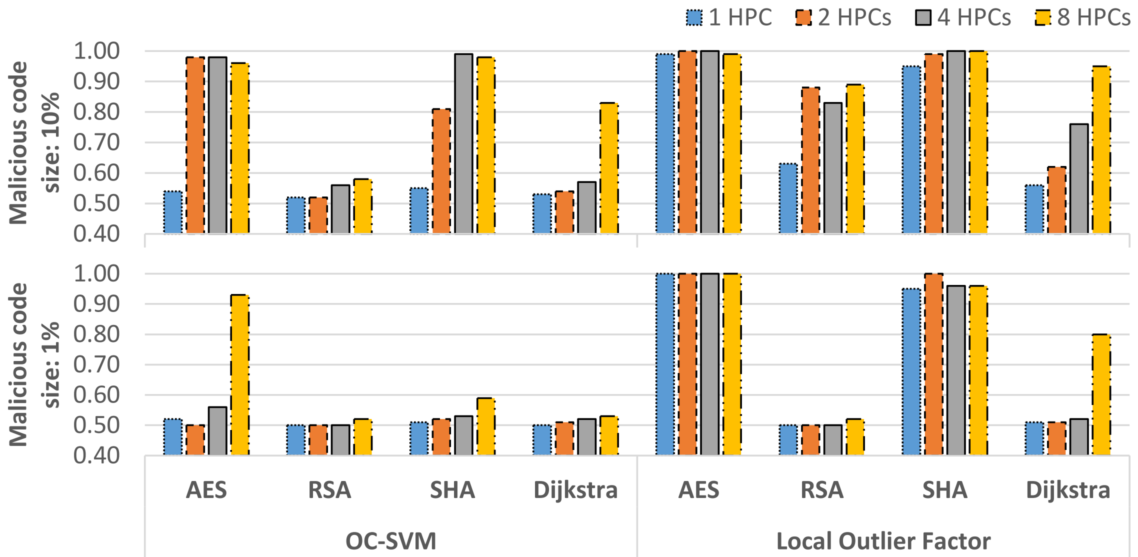
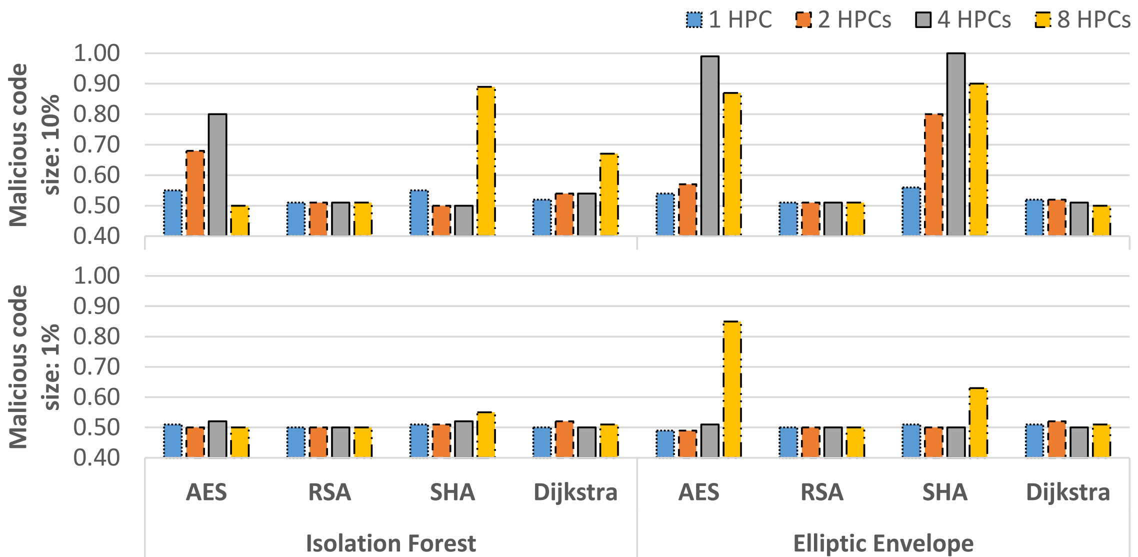


Figure 4.13: Autoencoder architecture for eight features.

This brief exploration of autoencoders in HMD did not demonstrate any improvement in the performance of hardware-based detectors. However, this may be a starting point for future studies, considering the several model configuration parameters that may be explored. Moreover, the detection performance increase may be more easily verified when analyzing datasets with many features (HPCs).



(a)



(b)

Figure 4.14: Accuracy when the autoencoder is inserted in the framework, for Stealthy mode. (a) OC-SVM and LOF classifiers. (b) Isolation Forest and Elliptic Envelope classifiers.

Chapter 5

Zero-day hardware-based detection of malware in operating systems

The content of this chapter is derived from the study conducted in collaboration with Bruno Davide as part of his master's research, "Malware detection using hardware performance counters on RISC-V based cloud servers," [185].

Operating systems are a collection of software and firmware that enable the execution of computer applications. They provide essential functions for booting, resetting, process management, memory handling, file system operations, and device control. OSs allow users to execute a multitude of tasks, such as creating and editing documents, browsing the internet, sending and receiving emails, listening to music and watching videos, playing games, organizing and store files, learning and studying, designing projects, printing documents, and many others. However, its broad complexity and capability open a large attack surface frequently exploited by malware.

Figure 5.1 shows the amount of new malware by year for **Microsoft Windows**, **Linux**, and **Android** OSs, according to the Av-Test Institute [186]. The widespread prevalence of Windows in personal computing continues to make it the main target for malicious software. However, malware targeting Linux and Android (the world's most widely used mobile operating system, based on Linux) has increased significantly over the past fifteen years. This trend is driven by the exponential popularity of mobile and embedded devices, propelled by the IoT. Mobile and embedded devices are built upon a variety of CPU architectures, often on hardware with limited resources, favoring the adoption of different flavors of Linux OS [187].

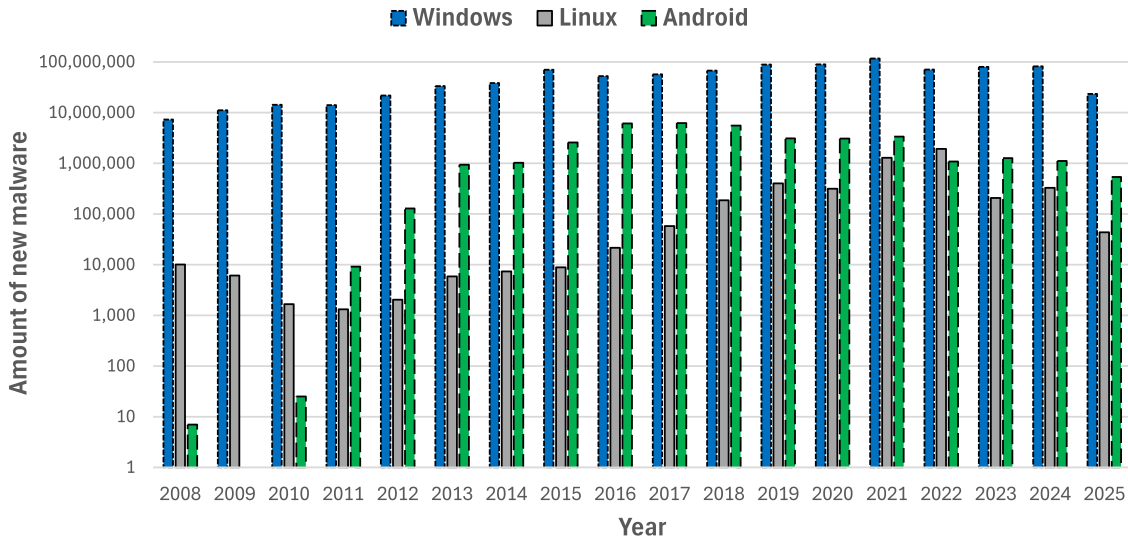


Figure 5.1: Amount of new malware by year for Windows, Linux, and Android [186].

This chapter's case study explores the performance of hardware-based approaches for zero-day detection of malware running in OSs, focusing on a Linux over RISC-V environment. The gem5 simulator was employed to simulate the whole system, a powerful tool encompassing system-level architecture and processor microarchitecture. Section 5.1 discusses the gem5's fundamental features. Section 5.2 outlines the experimental methodology. Finally, Section 5.3 details the setup and discusses the experimental results.

5.1 gem5 simulator

gem5 [143, 188] is a cycle-level event-driven computer architecture simulator. It results from the combined efforts of many academic and industrial institutions, including AMD, ARM, HP, MIPS, Princeton, MIT, and the Universities of Michigan, Texas, and Wisconsin. The availability is guaranteed by a license based on Berkeley Source Distribution (BSD), friendly to both industrial and academic institutions. Figure 5.2 shows an overview of gem5's architecture.

The implementation of gem5 follows key aspects suitable for designing simulators: object-oriented design, Python integration, domain-specific languages, and standard interfaces. The **object-oriented design** provides flexibility through independent and composable simulation components called SimObjects. They model concrete hardware components such as processor cores, caches, interconnect elements and devices, and more abstract entities like a workload and associated process context for system-call emulation. Every SimObject is represented by two

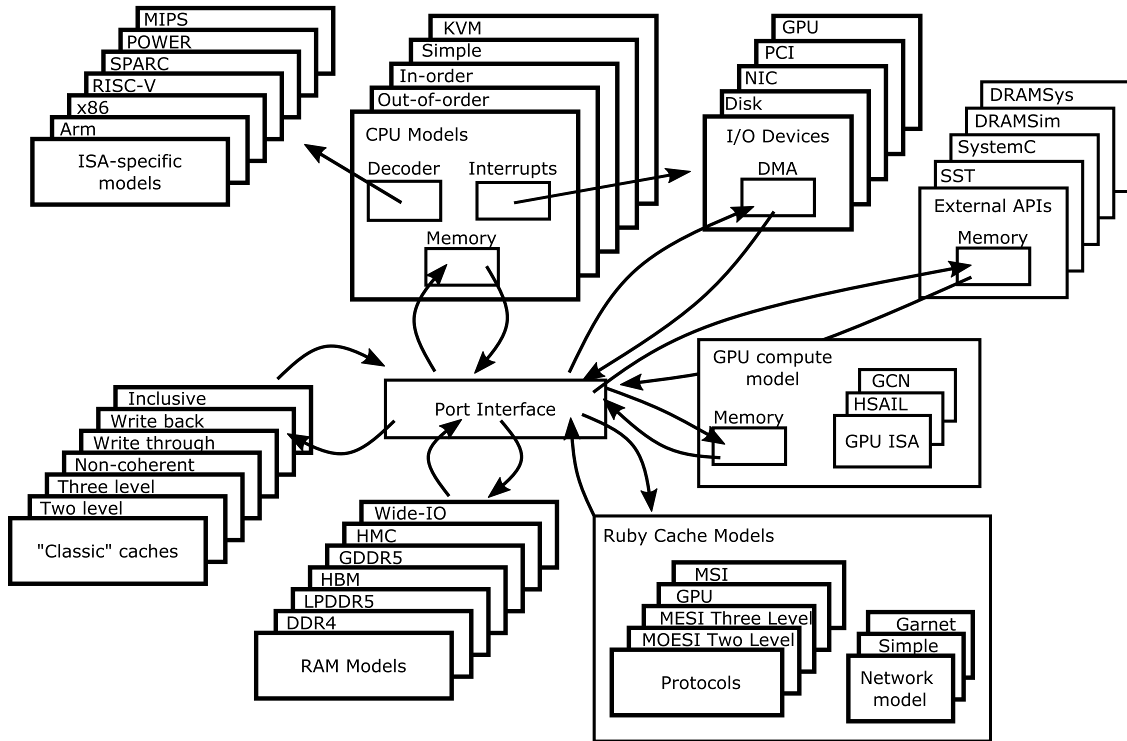


Figure 5.2: Overview of gem5's architecture [188].

classes, one in Python (for initialization, configuration, and simulation control) and one in C++ (for object state and remaining behavior). This **Python integration** delivers significant power to the simulator. **Domain-specific languages** refer to performing specialized tasks using languages common to the task's problem space. gem5 provides two domain-specific languages, one for specifying instruction sets (C++ base class) and one for specifying cache coherence protocols (Specification Language including Cache Coherence (SLICC)). **Standard interfaces** are fundamental for object-oriented design. Two central interfaces in gem5 are the port interface (used to connect two memory objects) and the message buffer interface (used internally in the memory system).

When simulating a system with gem5, some key options cover a wide range of speed versus accuracy trade-offs: CPU model, execution mode, and memory system. Four **CPU models** are provided: (i) "simple" is a minimal single instruction per cycle, offering the lowest overhead and suited for cases where a detailed model is not necessary. Generally used for fast-forwarding or cache-warming. Notably, the simple model is broken up into two specific types: "atomicSimple" which uses atomic memory accesses, and "timingSimple" which uses timing memory accesses; (ii) "minor" (also called "in-order") is an execute-in-execute CPU (instructions are only executed in the execute stage after all dependencies have been resolved) with an in-order pipeline; (iii) "O3" (also called "out-of-order") is an execute-in-execute

CPU with an in-order pipeline, being the highest configurable CPU model. When using minor or O3, the simulator runs slower but can provide more realistic results; (iv) "Kernel Virtual Machine (KVM)" mode bypasses simulation and allows the binaries running in `gem5` to use the underlying host's processor if the host ISAs is the same as the application in the simulator. It is based on the KVM API in Linux, performing nearly the same as when running natively. The simulator can operate in two **execution modes**, "system-call emulation" and "full-system", according to the Figure 5.3. In system-call emulation, `gem5` avoids the need to model devices or an OS by emulating most system-level services. In another way, full-system mode executes user- and kernel-level instructions and models a complete system, including the OS and devices. This last improves the simulation accuracy and the variety of workloads that `gem5` can execute. Regarding the **memory system**, the simulator provides "classic" and "ruby" options. The classic is a fast and easily configurable memory system, while the ruby is a flexible infrastructure capable of accurately simulating various cache-coherent memory systems.

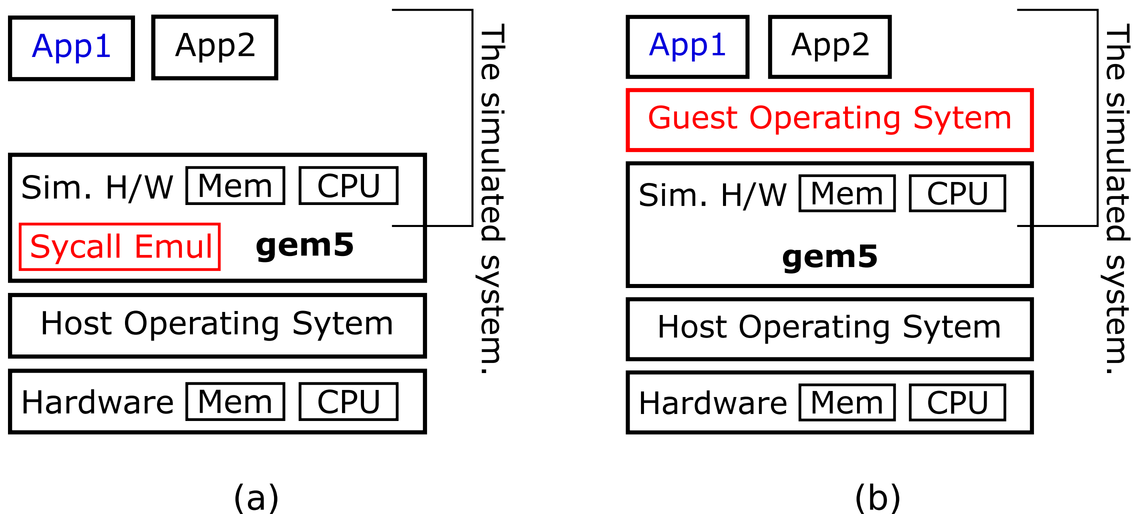


Figure 5.3: `gem5` execution modes [188]. (a) System-call emulation mode. (b) Full-system mode.

`gem5` currently supports several ISAs, including the dominants X86, Arm, and RISC-V and others like MIPS, Power, Scalable Processor Architecture (SPARC) and Graphics Processing Unit (GPU) ISAs [188]. These ISAs include the details to execute each instruction as well the system-specific devices necessary for full system simulation (Direct Memory Access (DMA), Universal Asynchronous Receiver-Transmitter (UART), interrupt, disk and Ethernet controllers, Peripheral Component Interconnect (PCI) components, and many others). The hardware for full-system execution mode is based on the HiFive platform from the American company SiFive [189] (Figure 5.4). The `gem5` software layers are shown in Figure 5.5. The CPU employed is the RISC-V RV64GC [190, 191].

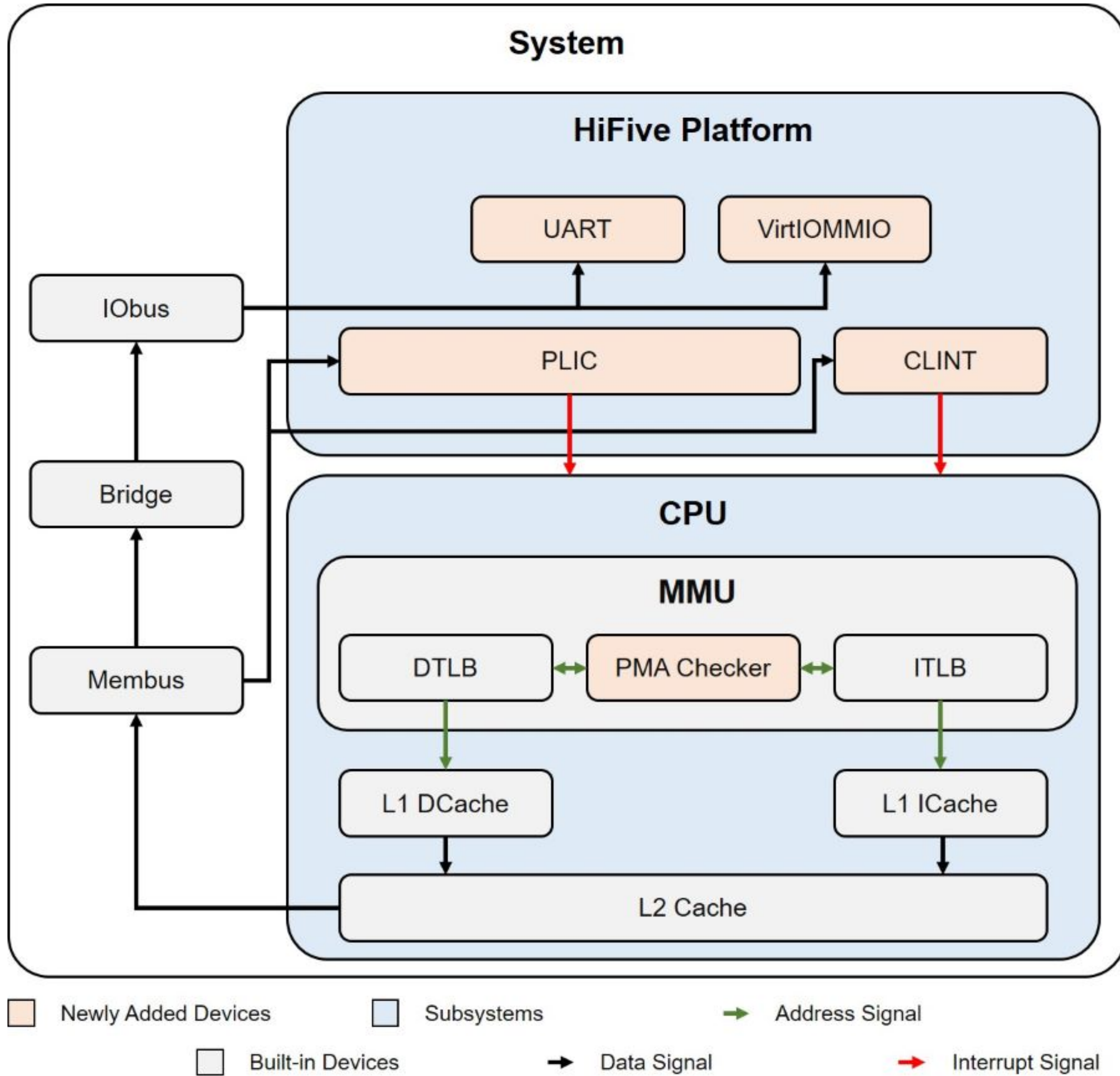


Figure 5.4: `gem5` hardware with RV64GC for full-system execution mode, based on HiFive platform [189].

5.2 Methodology

Figure 5.6 presents the pipeline to evaluate the effectiveness of HMD in detecting malware running in OSs. As can be seen, it has some similarities with the methodology in Chapter 4. The **simulation environment** (red box) employs the `gem5` simulator to run applications on a RISC-V core and collect the HPCs. The coupled **anomaly HMD framework** (green box) implements the hardware detection approaches targeting malware in OSs. As a crucial point of the methodology, also in this case study, the analysis technique used was sampling the HPCs only at the end of the application's execution.

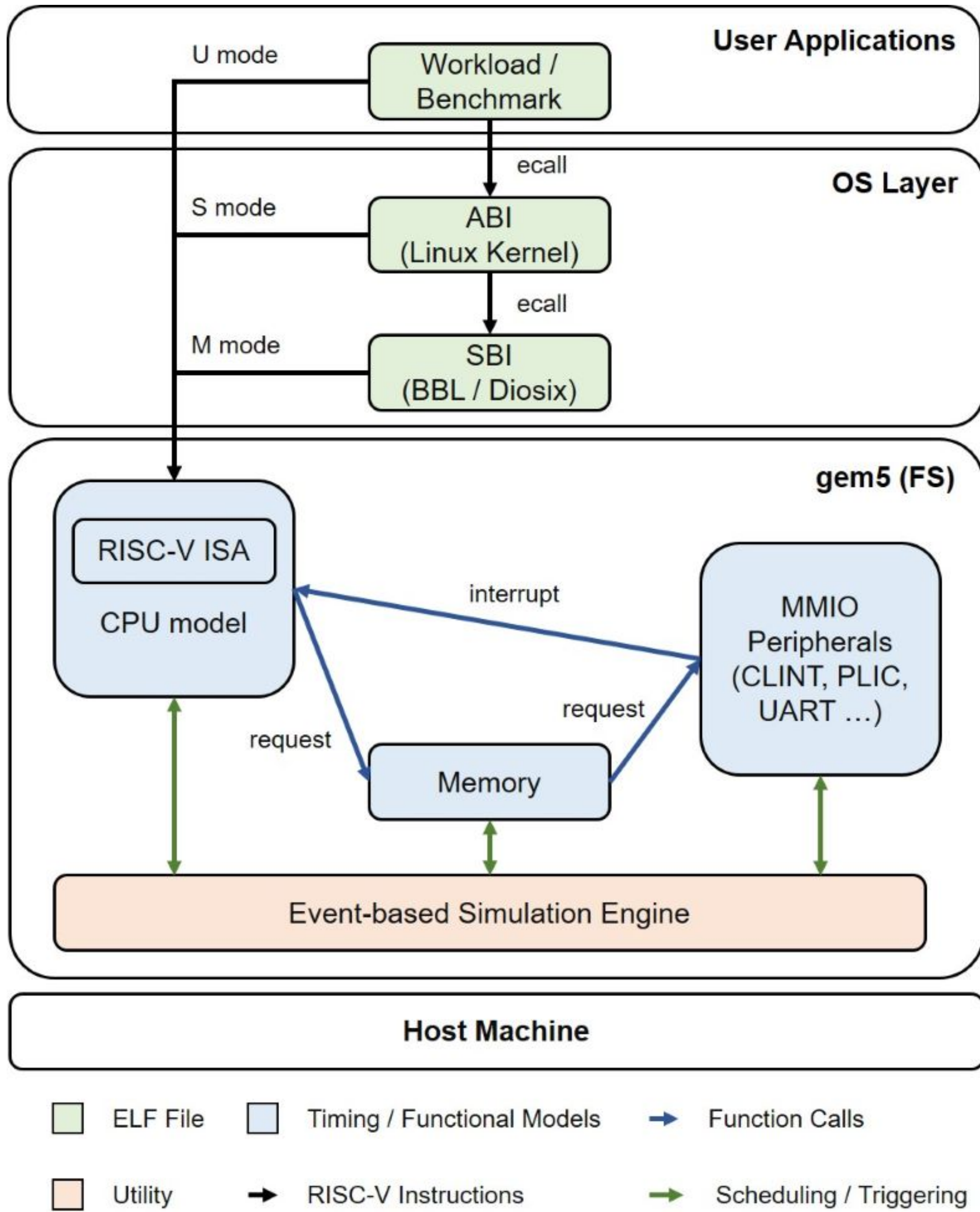


Figure 5.5: `gem5` software layers with RV64GC for full-system execution mode [189].

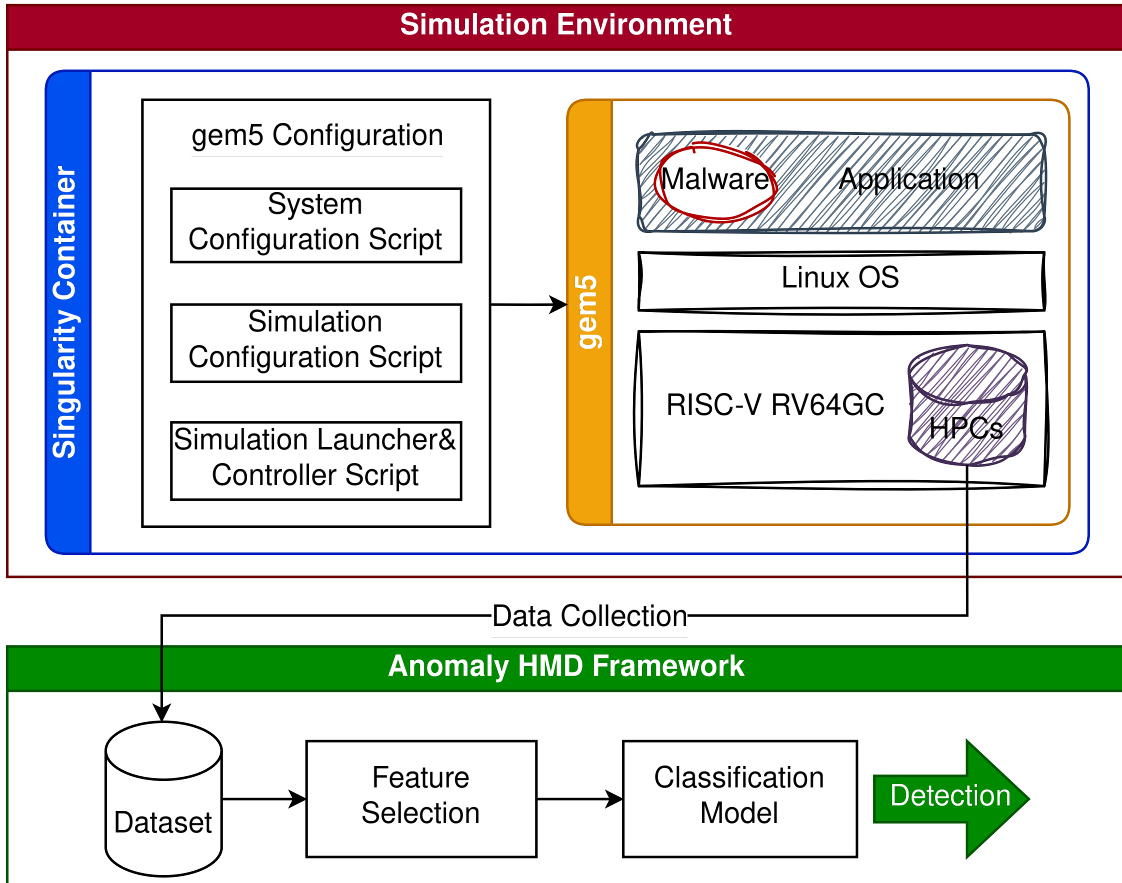


Figure 5.6: Methodology overview for the zero-day detection of malware in operating systems.

5.2.1 Simulation environment

The simulation environment runs in a **Singularity container** for speed up execution. It comprises the configuration scripts and the whole system simulator (`gem5`), which runs the applications over a RISC-V core and a Linux OS. The configuration scripts are fundamental pieces in the `gem5` simulation: they create the system to simulate, create all the system components, specify all the parameters for the system components, and begin the simulation. This study split them into three: the system configuration script, the simulation configuration script, and the simulation launcher&controller script.

The **system configuration script** creates the RISC-V system, with all the functional components that define its architecture. The CPU is created, and the platform is set up to HiFive, according to Figure 5.4. The CPU can be configured as simple or O3 models. This possibility allows us to use the faster simple model until the system boots up, saving the system's state using a checkpoint. Later, the

benchmarks can run in a more accurate model, switching to O3 and restoring the system's state from the checkpoint without the need to boot up the system again. The HPCs are available only when using the O3 accurate model. The system configuration script still creates and connects the cache hierarchy, Memory Management Unit (MMU), memory bus (Membus), and interrupt controller (Core Local Interrupt Controller (CLINT)). It also defines the clock and voltage domains. Finally, the bridge, MMU and Virtual Input-Output (VirtIO) are initialized. The VirtIO is part of the Virtual Input-Output & Memory-Mapped Input-Output (VirtIOMMIO) block, which provides a copy-on-write root file system based on a BusyBox image. This root file system includes basic Unix utilities and a minimal environment for running a full OS and applications. OS boot is handled by the RISC-V Proxy Kernel, which embeds the Berkeley Bootloader (BBL). This last wraps the statically compiled Linux kernel as a payload, producing a bootable binary that enables the kernel to start execution within the `gem5` simulation environment.

The **simulation configuration script** instantiates the RISC-V system created, allowing its simulation. It specifies essential parameters for the simulation, such as the RISC-V BBL, disk image, and the number and model of CPU to be used. Additionally, it defines a .rcS Linux file (a shell script executed when the system boots up), which we use to provide input parameters for the applications. Running this .rcS script initializes the simulation and restores checkpoints from the designated .cpt folder. The **simulation launcher&controller script** manages the automated execution and monitoring of applications. It performs the following procedures: (i) event selection, configuring which hardware events are monitored by the HPCs; (ii) HPCs initial reading; (iii) benchmark execution; (iv) post-execution HPCs reading, enabling to calculate the difference from the initial reading; and (v) output file generation, saving the HPCs values in a file. The benchmark execution has two modes (as illustrated in Figure 5.7): (i) **Benign**, where just a benign application is executed; (ii) and **Malign**, where a malware is executed after the benign application.

5.2.2 Hardware-based malware detection framework

The HMD framework is similar to the one in Chapter 4, consisting of feature extraction, feature selection, classification model training, and performance assessment. The **feature extraction** is performed by the function `read_csr_safe` called in the script simulation launcher&controller after the execution of the application (or application plus malware). The HPCs are accessed via the thread context of the simulation process, as in `gem5`, the counters available on real hardware are not implemented. Two different datasets are generated with the data extracted: (i) benign for training and (ii) malignant for testing, comprising both application and malware data executions in the latter.

Like Chapter 4, the **feature selection** comprised (i) a manual pruning on the

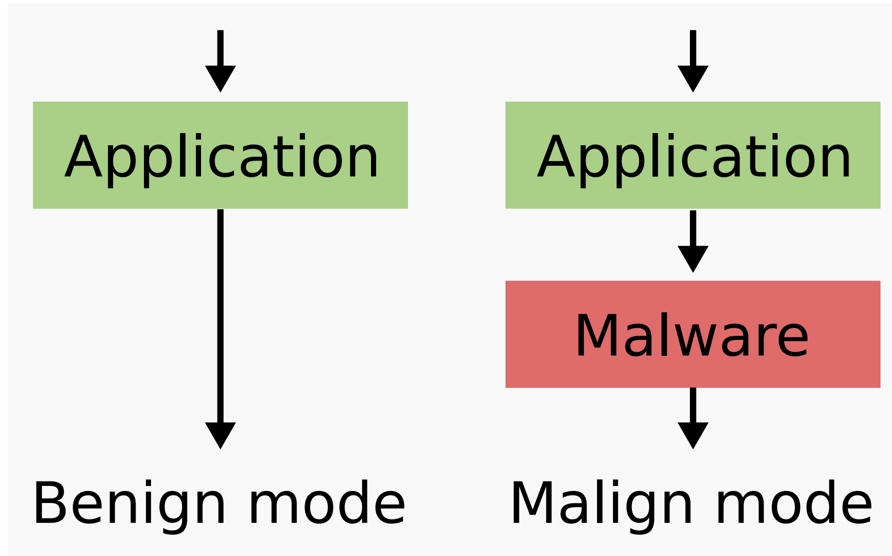


Figure 5.7: Benchmark execution modes.

complete list of extracted HPCs and (ii) a feature ranking elaboration based on the PCA technique. The manual pruning removed inactive (zero-locked) HPCs, and counters that the literature states which have limited value in providing insights into program behavior [90, 94]: `ELAPSED CYCLES`, the total number of cycles; `ELAPSED TIME`, the total execution time; and `ELAPSED INSTRUCTIONS`, the total number of instructions executed. Next, the feature ranking elaboration used the PCA technique [176, 177] to rank the remaining HPCs. This two-step feature selection ranks HPCs by priority, ensuring a broad characterization of the detector related to the number of features employed, facilitating the comparison in future analysis. As **classification models**, remaining with the anomaly detection technique, the following four algorithms were used: OC-SVM, LOF, Isolation Forest, and Elliptic Envelope.

5.3 Results

Figure 5.8 shows the setup for simulations of zero-day malware detection in operating systems. The experiments are executed in a **host computer** with processor AMD Ryzen 9 7950X and OS Ubuntu 22.04.3 LTS (Jammy Jellyfish). A **Singularity container** also running Ubuntu 22.04.3 packages the setup. The **`gem5` simulator** is executed in this environment. It simulates the behavior of a **HiFive platform** [189] with the **RISC-V RV64GC** core. This CPU is a 64-bit processor that implements standard extensions (IMAFD codes) and compressed instructions. 31 HPCs are modeled by the `gem5` simulator. The simulated hardware runs a **Linux OS (kernel 5.18)**, with **RISC-V BBL 5.1**.

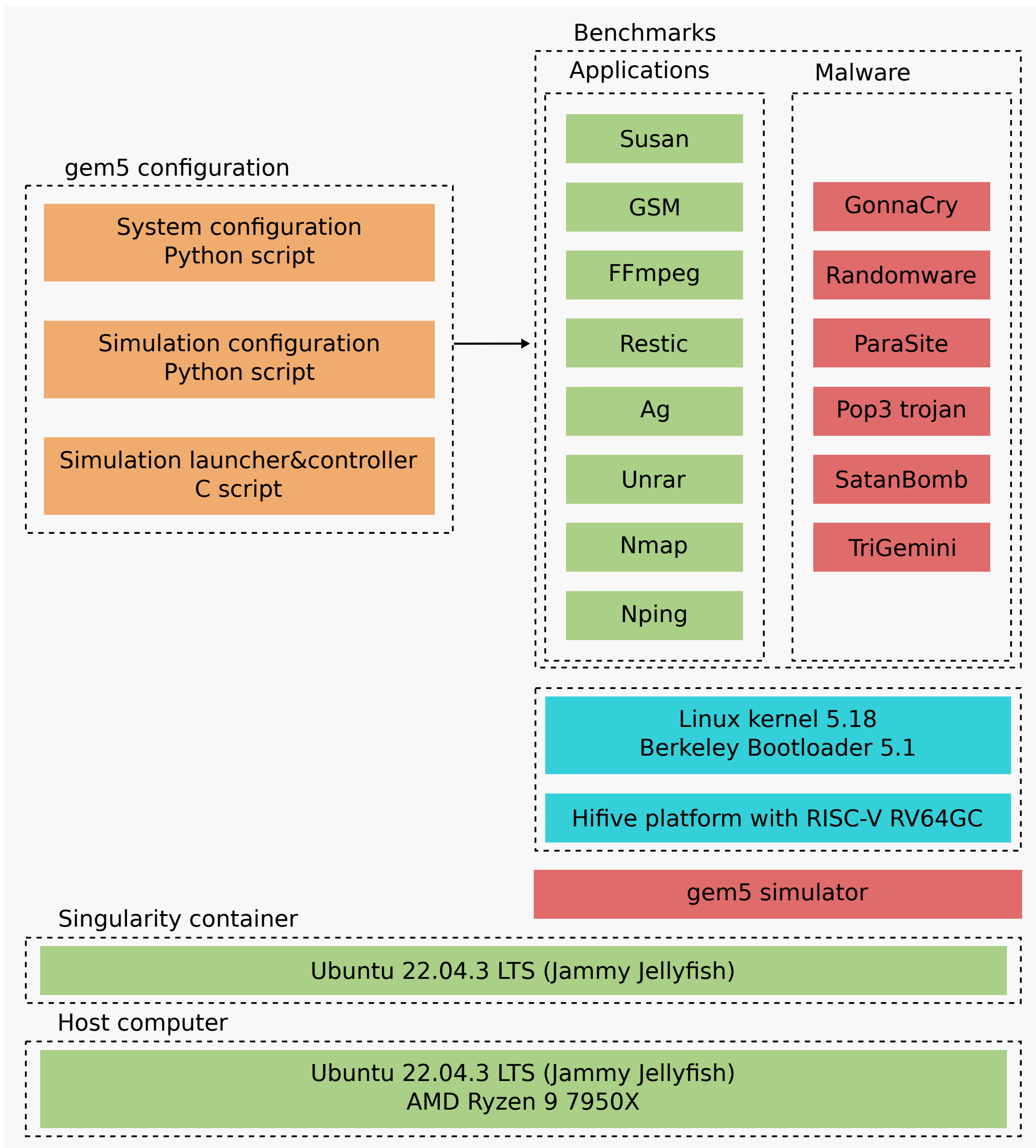


Figure 5.8: Setup for simulations of zero-day detection of malware in the operating system.

The benchmarks comprise eight applications and six malware, detailed in Table 5.1. They are free and open-source codes cross-compiled using the RISC-V GNU Toolchain (GNU means "GNU is not Unix"). Most malware examples were created by the authors for educational and research purposes. The selection of these specific applications and malware was guided by the following requirements:

Linux compatibility, source code availability, reduced dependencies for successful cross-compilation, and consistent behavior across various input stimuli. Moreover, the chosen malware benchmarks represent a range of goals and attack scenarios. Likewise, the selected applications cover commonly used program tasks and behaviors, including compression and decompression, encoding and decoding, audio and image processing, internet connection stress, and searching tools.

Table 5.1: Benign applications and malware employed as benchmarks.

Type	**Name**	**Description**	**Ref.**
Benign app.	Susan	It is an image-processing application that recognizes corners and edges. Susan works by examining pixel neighborhoods and comparing brightness values, typically employed with grayscale images	[165]
Benign app.	GSM	It is a voice encoder and decoder based on the Global System for Mobile Communications (GSM) standard. GSM employs a combination of Time Division Multiple Access (TDMA) and Frequency Division Multiple Access (FDMA) to process the data streams	[165]
Benign app.	FFmpeg	It is an application to convert multimedia files (such as audio, video, and other) between formats	[192]
Benign app.	Restic	It is an application to back up files, providing encrypted, deduplicated, and incremental backups. The storage can be locally or remotely	[193]
Benign app.	Ag	Also known as The Silver Searcher, it is a search application designed as an alternative to `ack` and `grep` tools. Ag is optimized for searching code repositories and large text files	[194]
Benign app.	Unrar	It is an application for extracting .rar file archives	[195]
Benign app.	Nmap	It is an application for network discovery and security auditing. Nmap uses raw IP packets to determine what hosts are available on the network, what services those hosts are offering, what OSs are running, what type of packet filters/firewalls are in use, and dozens of other characteristics	[196]
Benign app.	Nping	It is an application for network packet generation, generating network packets for many protocols. Nping can be used as a simple ping utility, a packet generator for stress testing, Address Resolution Protocol (ARP) poisoning, DoS attacks, route tracing, etc.	[197]
Malware	GonnaCry	It is a ransomware that encrypts files with AES encryption	[198]
Malware	Randomware	It is another ransomware that encrypts files with a simple XOR	[199]
Malware	ParaSite	It is a backdoor that allows a server to connect to infected remote clients and replicate itself on the client	[200]
Malware	Pop3 trojan	It is a trojan that enables an attacker to have control of a remote computer through a malicious Post Office Protocol 3 (POP3) daemon (the trojan)	[200]
Malware	SatanBomb	It is a botnet designed to make DoS attacks. It uses a fork function to continually replicate a process and deplete available system resources, slowing down or crashing the system	[199]
Malware	TriGemini	It is a botnet capable of launching various types of distributed DoS attacks. It supports multiple attack vectors such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Internet Control Message Protocol (ICMP)	[201]

Once the setup is operational, for each application, we conducted a specific number of executions, as presented in Table 5.2. Half of the executions are for training and half for testing, balancing proper detector model evaluation and training effectiveness. Executions for training are just in benign mode (see Figure 5.7). Executions for testing are 50% in benign mode and 50% in malign mode. In this last, the six malware are executed after each application, randomly drawn from

each execution. It is ensured that the malware portion is evenly represented across all six malicious applications, preventing any single malware from dominating the dataset. In each execution, the simulation configuration script varies the application input stimulus, enabling attack detection regardless of the application input. Within the malware set, only the input of TriGemini varies (in timing, protocols, and flooding rate).

Table 5.2: Number of simulations executed.

Application	Training set	Test set	
	100% benign mode	50% benign mode	50% malign mode
Susan	10000	5000	5000
GSM	200	100	100
FFmpeg	550	275	275
Restic	200	100	100
Ag	350	175	175
Unrar	150	75	75
Nmap	250	125	125
Nping	150	75	75

Similar to Chapter 4, the feature selection and the classification models in the HMD framework are implemented using the `Scikit-learn` library for Python. As a performance evaluation metric, only accuracy is shown for simplification. But precision, recall, specificity, F1-score, and AUC were also evaluated and did not reveal any novelty related to the accuracy.

Analyzing the datasets generated, we observe a small number of executions with HPCs values as negative. We attributed these values as an overflow of the counters, considering that the current version (1.13) of RISC-V privileged specification [136] does not specify a mechanism to generate an interrupt when the counters overflow. To avoid this loss of control of interfering in the analysis, we chose to remove from the dataset these executions with negative HPC values.

5.3.1 Feature selection

Table 5.3 presents the result of feature selection, showing the HPCs remained after manual pruning, ranked according to PCA. It is worth noticing that the `number of instructions committed` is the most significant HPC for all the applications. The `number of L1 data cache accesses` and the `number of total data memory access` are for several applications, the second and third most significant HPCs, indicating that memory access patterns are highly informative. The following analysis presents the detection performance as a function of the number of HPCs, prioritizing them in order of significance.

Table 5.3: Result of feature selection, with HPCs remained after manual pruning and ranked according to PCA. Index 1 defines the most significant counter.

HPC	**Susan**	**GSM**	**FFmpeg**	**Restic**	**Ag**	**Unrar**	**Nmap**	**Nping**
Number of instructions committed	1	1	1	1	1	1	1	1
Number of L1 data cache accesses	2	3	4	2	2	2	2	4
Number of total data memory access	3	4	5	3	3	3	3	5
Number of load instructions	4	2	3	9	5	4	4	2
Number of branch prediction lookups	5	5	8	4	4	5	5	3
Number of control flow instructions	6	8	7	5	6	7	7	6
Number of L1 instruction cache access	7	7	2	6	7	6	9	7
Number of store instructions	8	6	6	8	9	9	6	9
Number of conditional branches predicted	9	9	10	7	8	8	8	8
Number of L1 data cache misses	10	13	9	10	11	10	11	13
Number of branch mispredictions	11	10	11	13	12	11	12	12
Number of L1 data cache write-backs	12	18	14	19	16	14	17	21
Number of L1 instruction cache misses	13	14	16	14	13	18	21	15
Number of L2 cache misses	14	11	12	12	10	12	10	10
Number of data TLB misses	15	19	17	17	17	15	14	18
Number of data TLB miss write	16	23	20	15	21	19	18	24
Number of L2 cache write-backs	17	15	13	18	14	13	13	11
Number of data TLB miss read	18	20	18	22	18	17	15	19
Number of exceptions	19	17	19	11	19	20	19	16
Number of branch misfetches	20	16	21	16	20	21	20	17
Number of L2 cache hits	21	12	15	20	15	16	16	14
Number of instruction TLB misses	22	24	22	21	22	22	22	20
Number of exception return instructions	23	22	24	24	24	24	24	23
Number of pipeline flushes due to mispredictions or exceptions	24	21	23	23	23	23	23	22

5.3.2 Performance

Figure 5.9 shows the accuracy in detecting malware in applications running in the operating system, with data clustered by application. The six bars for each application and classifier refer to performance related to 1, 2, 4, 8, 16, and 24 HPCs (from left to right in the figure, respectively). When using a single HPC, it is employed as the most significant counter (index 1) in Table 5.3. When using two, the first and second most significant counters (index 1 and 2), and so on.

Analyzing the overall detection performance by application, and considering the best case, we have four applications with accuracy higher than 90% (Ag 99%, Susan 97%, Nping 96%, and GSM 91%), two applications with accuracy higher than 80% (Unrar 88% and Restic 83%), FFmpeg with accuracy of 70% and Nmap with accuracy of 68%. We may observe that the performance is highly dependent on the application since the delta accuracy is almost 30% (from 97 to 68%). The different number of executions/samples between the applications does not affect the performance results. For instance, Nping and GSM, in the group with accuracy higher than 90%, have fewer samples than FFmpeg and Nmap, the worst performances.

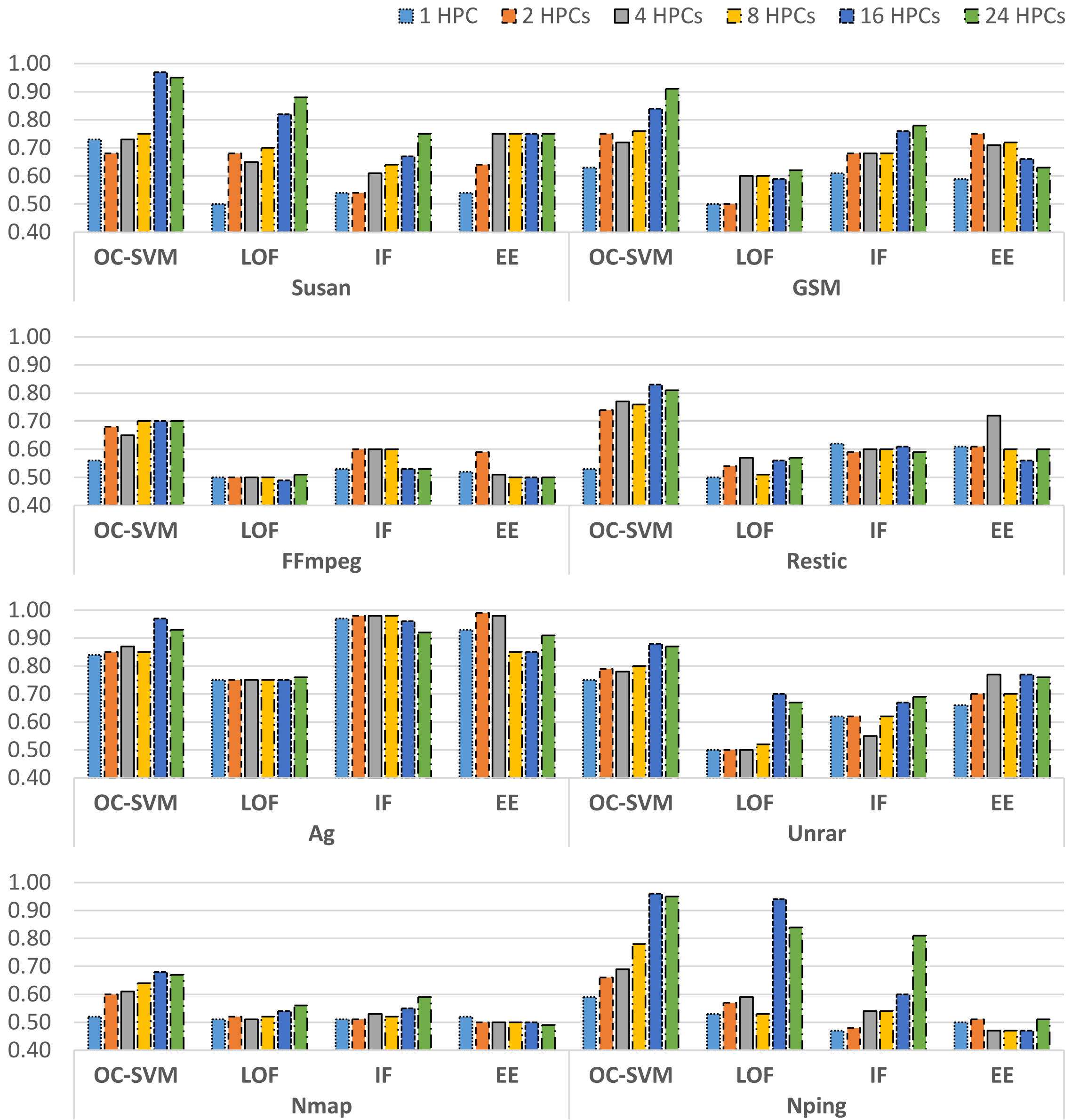


Figure 5.9: Accuracy in detecting malware in applications running in the operating system (clustered by application). IF is an Isolation Forest, and EE is an Elliptic Envelope.

It is worth noticing that almost all of the best cases mentioned used the OC-SVM classifier (the exception is the Ag, which performed better with the Elliptic Envelope). This prowess is evident in Figure 5.10, which contains the same data as Figure 5.9 but is clustered by the classifiers. We should still highlight that in the best cases with OC-SVM, 6 out of 7 applications employ 16 HPCs instead

of the maximum 24 HPCs analyzed. It indicates that, for our analysis with OC-SVM, 16 is close to the optimum number of HPCs. Regarding the remaining classifiers, no single model outperformed the others significantly. LOF performed better with Susan and Nping; Isolation Forest with GSM, FFmpeg, and Nmap; and Elliptic Envelope with Restic, Ag, and Unrar. Conversely, the overall performance of these three classifiers is low, exceeding 80% accuracy with few applications: LOF in two applications (Susan and Nping); Isolation Forest in two applications (Ag and Nping); and Elliptic Envelope just in one application (Ag). Regarding the number of HPCs analyzed, OC-SVM and LOF significantly improve accuracy when going from few to many counters. However, Isolation Forest and Elliptic Envelope leverage less the increase in the number of features, with a highlight to this last one, which several times reduced the accuracy when considering more counters.

Finally, we also evaluated the capacity of training a single classifier to detect malware across all eight applications, similar to what was performed in Chapter 4. Overall, the accuracy was poor, making malware detection unfeasible. Once again, this suggests that using specialized detectors for each application is the most effective solution.

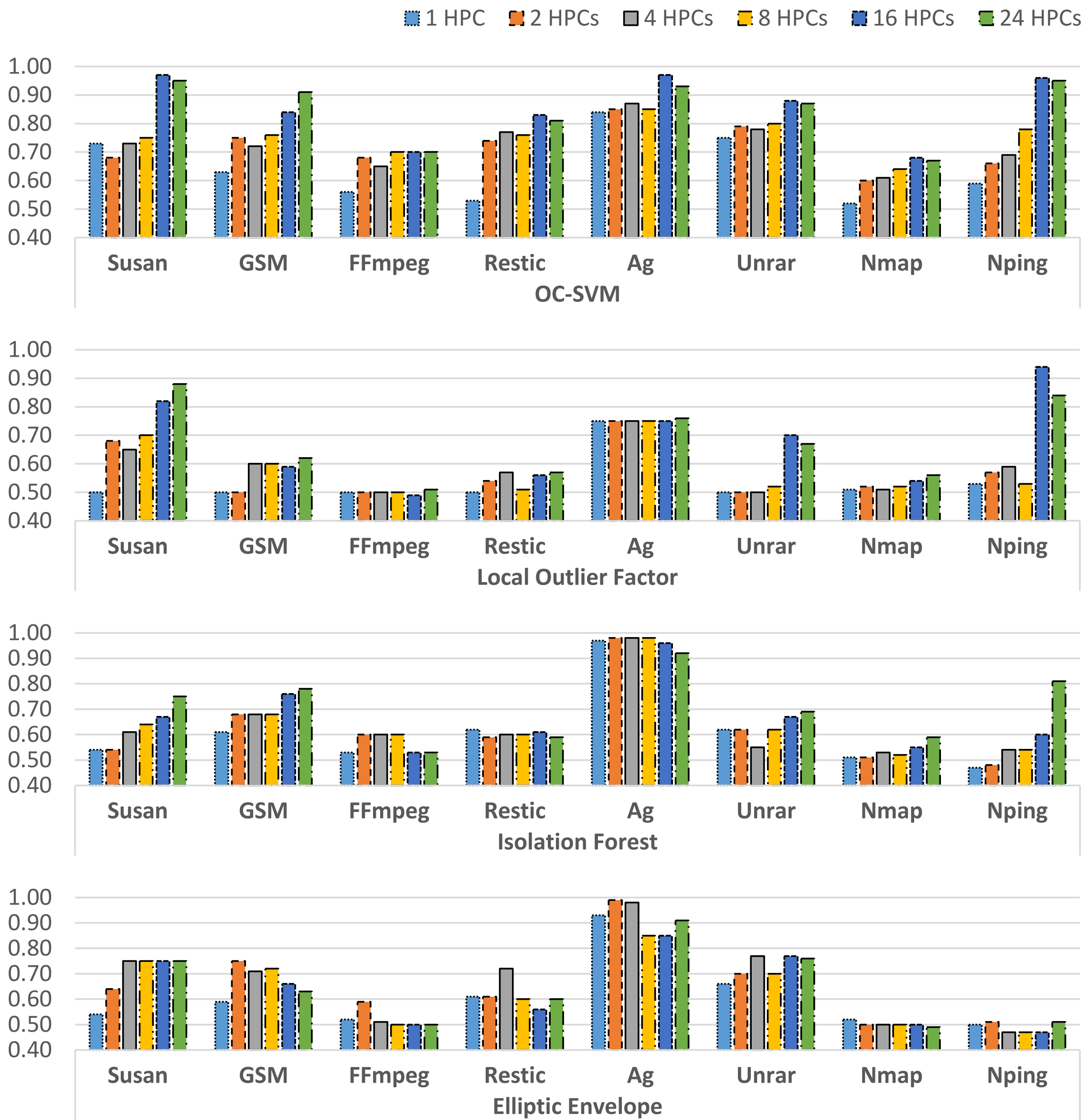


Figure 5.10: Accuracy in detecting malware in applications running in the operating system (clustered by classifier).

Chapter 6

Conclusions

Cyber insecurity is among the top global risks of the contemporary world, and malware is the primary vector for cybercrimes. This thesis delved into malware detection, the first step in promoting secure computers and cyber environments. HMD, the focus here, is the latest approach and consists of dynamically analyzing hardware events in a CPU, leveraging ML algorithms to distinguish between benign applications and malware. This thesis investigated this approach in depth, focusing on how the core component - the ML methods - should be employed to optimize performance and efficiency.

This thesis yielded several **findings** that may help with malware detection. **First -** although a single standard taxonomy is absent, a useful way to dissect malware is through its goals and three properties: vulnerability exploitation, propagation method, and concealment strategy. CVE and CWE lists are straightforward resources to deal with malware, assuring a worldwide recognized reference. **Second -** a precise comparison between malware detection approaches enables the identification of the advantages of HMD precisely. The list comprises the ability for runtime, zero-day, and stealthy malware detection, resilience against subverting the protection, low-performance overhead, and reduced detection cost. **Third -** based on a generic hardware-based detection framework, the ML methods and strategies frequently employed in HMD studies were discussed. They included libraries, utilities, simulators, and sampling methods for feature extraction; algorithms, libraries, and common practices for feature selection; techniques (like anomaly detection), algorithms, and libraries for classifiers. Additionally, a section is dedicated to identifying ML methods explored by recent studies to improve performance, such as ensemble learning, specialization, adaptive detection, and time series.

Fourth - the case studies of Chapters 4 and 5 investigated the HMD performance capabilities in detecting zero-day malware (specifically, SBO attacks and malware running in OSs). Overall, they showed that anomaly HMD can detect zero-day threats, but that accuracy is a significant challenge. Implementations must be carefully performed to overcome poor/unfeasible detection, and points requiring

attention include: (i) classifiers should be tailored to specific applications, i.e., detectors should be dedicated by application; (ii) the detection performance is highly dependent on the application, with randomness/unpredictability of protected applications an obstacle; (iii) classification algorithms have specific performances in different scenarios (there is no best algorithm), and to match the algorithm with the application is a must. (iv) an optimal number of ML features (HPCs) helps tune performance. Too little information offers insufficient insight into the software, while too much can hinder it. **Fifth -** the two case studies presented focused on RISC-V, showing that this platform may improve its PMU specification to better support HMD approaches. For instance, the number of hardware events leveraged in the CV32E40p RISC-V core is small for proper software profiling. Furthermore, the counter overflow reported in `gem5`, caused by the lack of a specified mechanism for generating interrupts in the current version (1.13) of the RISC-V privileged specification, may lead to errors in malware detection.

To conclude this thesis that registered a journey about malware detection (mainly using ML), some paths are pointed out for **future research** on the field: (i) the randomness/unpredictability of protected applications is a significant obstacle in the performance of HMD, understand how to treat it is a step ahead; (ii) a promising solution for malware detection combine the software and hardware approaches, with hardware as the primary defense. This solution's design may maximize each approach's strengths, generating effective and lightweight detectors; (iii) like mixing software approach with events from the architecture and microarchitecture, HMD may also leverage silicon sensor data to improve accuracy. For example, future studies may add temperature, current, and voltage to have insights from the software; (iv) Malware Runtime Detection (MRD) (a novel acronym) is a top desire in the field. The design, implementation, and testing of hardware modules dedicated to security is valuable research. These modules may include hardware accelerators for ML and several additional features.

Chapter 7

List of publications

1. C. P. Chenet, A. Savino, and S. Di Carlo, “A survey on hardware-based malware detection approaches,” IEEE Access, vol. 12, pp. 54 115–54 128, 2024.

2. C. P. Chenet, A. Savino, and S. Di Carlo, “Zero-day hardware-supported malware detection of stack buffer overflow attacks: an application exploiting the cv32e40p risc-v core,” in 2025 IEEE 26th Latin American Test Symposium (LATS), 2025, pp. 1–6.

3. C. P. Chenet, A. Savino, and S. Di Carlo, “Using analog scrambling circuits for automotive sensor integrity and authenticity,” in 2022 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2022, pp. 1–6.

4. M. Alonso et al., “Special session: Security and ras in the computing continuum,” in 2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2024, pp. 1–6.

5. M. Alonso et al., “Validation, verification, and testing (vvt) of future risc-v powered cloud infrastructures: the vitamin-v horizon europe project perspective,” in 2023 IEEE European Test Symposium (ETS), 2023, pp. 1–6.

6. R. Canal et al., “Vitamin-v: Virtual environment and tool-boxing for trustworthy development of risc-v based cloud services,” in 2023 26th Euromicro Conference on Digital System Design (DSD), 2023, pp. 302–308.

Chapter 8

Code availability

In accordance with the principles and values of the open-source philosophy, and to support transparency, reproducibility, and collaboration within the scientific community, the developed code has been made publicly accessible as follows:

1. **Zero-day hardware-based detection of stack buffer overflow attacks:** https://github.com/smilies-polito/ZeroDay_HWBasedMalwareDetection_SBOAttacks/releases/tag/v1.0

Bibliography

[1] Cybersecurity Ventures. *2024 Cybersecurity Almanac: 100 Facts, Figures, Predictions And Statistics*. Available at `https://cybersecurityventures.com/cybersecurity-almanac-2024/` (2025/10/04).

[2] World Economic Forum. *The Global Risks Report 2023*. Available at `https://www3.weforum.org/docs/WEF_Global_Risks_Report_2023.pdf` (2025/10/04).

[3] Cybersecurity Ventures. *Cybercrime To Cost The World $9.5 trillion USD annually in 2024*. Available at `https://cybersecurityventures.com/cybercrime-to-cost-the-world-9-trillion-annually-in-2024/` (2025/10/04).

[4] Cybersecurity Ventures. *Global Cybersecurity Spending To Exceed $1.75 Trillion From 2021-2025*. Available at `https://cybersecurityventures.com/cybersecurity-spending-2021-2025/` (2025/10/04).

[5] François-Xavier Standaert. «Introduction to Side-Channel Attacks». In: *Secure Integrated Circuits and Systems*. Ed. by Ingrid M.R. Verbauwhede. Boston, MA: Springer US, 2010, pp. 27–42. ISBN: 978-0-387-71829-3. DOI: `10.1007/978-0-387-71829-3_2`. URL: `https://doi.org/10.1007/978-0-387-71829-3_2`.

[6] Associated Press News. *German hospital hacked, patient taken to another city dies*. Available at `https://https://apnews.com/article/technology-hacking-europe-cf8f8eee1adcec69bcc864f2c4308c94` (2025/10/04).

[7] Leyla Bilge and Tudor Dumitraş. «Before we knew it: an empirical study of zero-day attacks in the real world». In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 833–844. ISBN: 9781450316514. DOI: `10.1145/2382196.2382284`. URL: `https://doi.org/10.1145/2382196.2382284`.

[8] Faranak Abri et al. «Can Machine/Deep Learning Classifiers Detect Zero-Day Malware with High Accuracy?» In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 3252–3259. DOI: `10.1109/BigData47090.2019.9006514`.

[9] Zhangying He et al. «When Machine Learning Meets Hardware Cybersecurity: Delving into Accurate Zero-Day Malware Detection». In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 2021, pp. 85–90. DOI: 10.1109/ISQED51717.2021.9424330.

[10] Fatemeh Deldar and Mahdi Abadi. «Deep Learning for Zero-Day Malware Detection and Classification: A Survey». In: *ACM Comput. Surv.* 56.2 (Sept. 2023). ISSN: 0360-0300. DOI: 10.1145/3605775. URL: https://doi.org/10.1145/3605775.

[11] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*. Tech. rep. UCB/EECS-2016-129. EECS Department, University of California, Berkeley, July 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html.

[12] Varun Chandola, Arindam Banerjee, and Vipin Kumar. «Anomaly Detection: A Survey». In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[13] SonicWall. *2024 SonicWall Mid-Year Cyber Threat Report*. Available at https://www.sonicwall.com/threat-report (2024/11/14).

[14] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. 4th. Pearson Education, 2017.

[15] David Salomon. *Elements of Computer Security*. London: Springer London, 2010.

[16] IBM. *The history of malware: A primer on the evolution of cyber threats*. Available at https://www.ibm.com/think/topics/malware-history (2024/11/11).

[17] Anitta Patience Namanya et al. «The World of Malware: An Overview». In: *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2018, pp. 420–427. DOI: 10.1109/FiCloud.2018.00067.

[18] Fred Cohen. «Computer viruses: Theory and experiments». In: *Computers & Security* 6.1 (1987), pp. 22–35. ISSN: 0167-4048. DOI: https://doi.org/10.1016/0167-4048(87)90122-2. URL: https://www.sciencedirect.com/science/article/pii/0167404887901222.

[19] Ali Ahmed Mohammed Ali Alwashali, Nor Azlina Abd Rahman, and Noris Ismail. «A Survey of Ransomware as a Service (RaaS) and Methods to Mitigate the Attack». In: *2021 14th International Conference on Developments in eSystems Engineering (DeSE)*. 2021, pp. 92–96. DOI: 10.1109/DeSE54285.2021.9719456.

[20] Per Håkon Meland, Yara Fareed Fahmy Bayoumy, and Guttorm Sindre. «The Ransomware-as-a-Service economy within the darknet». In: *Computers & Security* 92 (2020), p. 101762. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.101762. URL: https://www.sciencedirect.com/science/article/pii/S0167404820300468.

[21] Grace Segers. *Cyberattack prompts major pipeline operator to halt operations.* Available at https://www.cbsnews.com/news/colonial-pipeline-cyberattack-shut-down/ (2025/03/17).

[22] Pratim Milton Datta and Thomas Acton. «Ransomware and Costa Rica's national emergency: A defense framework and teaching case». In: *Journal of Information Technology Teaching Cases* 14.1 (2024), pp. 56–67.

[23] Matt Bishop. *A taxonomy of (Unix) system and network vulnerabilities.* Tech. rep. Davis CA, 1995.

[24] Thomas A. Longstaff John D. Howard. *A Common Language for Computer Security Incidents.* Tech. rep. Livermore, CA, 1998.

[25] Daniel Lowry Lough and Nathaniel J. Davis. «A taxonomy of computer attacks with applications to wireless networks». PhD thesis. Blacksburg, VA, 2001.

[26] Simon Hansman and Ray Hunt. «A taxonomy of network and computer attacks». In: *Computers & Security* 24.1 (2005), pp. 31–43. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2004.06.011. URL: https://www.sciencedirect.com/science/article/pii/S0167404804001804.

[27] The MITRE Corporation. *The Common Vulnerabilities and Exposures Program.* Available at https://www.cve.org/ (2024/11/25).

[28] Maria Kjaerland. «A taxonomy and comparison of computer security incidents from the commercial and government sectors». In: *Computers & Security* 25.7 (2006), pp. 522–538. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2006.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0167404806001234.

[29] Nancy Gallagher Charles Harry. *Classifying Cyber Events: A Proposed Taxonomy.* Tech. rep. College Park, MD, 2018.

[30] Jelena Mirkovic and Peter Reiher. «A taxonomy of DDoS attack and DDoS defense mechanisms». In: *SIGCOMM Comput. Commun. Rev.* 34.2 (Apr. 2004), pp. 39–53. ISSN: 0146-4833. DOI: 10.1145/997150.997156. URL: https://doi.org/10.1145/997150.997156.

[31] Shankar Sastry Bonnie Zhu. *SCADA-specific Intrusion Detection/Prevention Systems: A Survey and Taxonomy.* Tech. rep. Berkeley, CA, 2017.

[32] Nils Gruschka and Meiko Jensen. «Attack Surfaces: A Taxonomy for Attacks on Cloud Services». In: *2010 IEEE 3rd International Conference on Cloud Computing*. 2010, pp. 276–279. DOI: 10.1109/CLOUD.2010.23.

[33] Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo. «A Survey on Hardware-Based Malware Detection Approaches». In: *IEEE Access* 12 (2024), pp. 54115–54128. DOI: 10.1109/ACCESS.2024.3388716.

[34] NIST. *Glossary*. Available at https://csrc.nist.gov/glossary (2024/11/14).

[35] ENISA. *Glossary*. Available at https://www.enisa.europa.eu/topics/incident-response/glossary (2024/11/14).

[36] Stuart Staniford, Vern Paxson, and Nicholas Weaver. «How to Own the Internet in Your Spare Time». In: *Proceedings of the 11th USENIX Security Symposium*. USA: USENIX Association, 2002, pp. 149–167. ISBN: 1931971005.

[37] Micha Moffie et al. «Hunting Trojan Horses». In: *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ASID '06. San Jose, California: Association for Computing Machinery, 2006, pp. 12–17. ISBN: 1595935762. DOI: 10.1145/1181309.1181312. URL: https://doi.org/10.1145/1181309.1181312.

[38] Federal Trade Commission Staff. *Monitoring Software On Your Personal Computer: Spyware, Adware, and Other Software*. Tech. rep. 2005.

[39] P. McFedries. «Technically Speaking: The Spyware Nightmare». In: *IEEE Spectrum* 42.8 (2005), pp. 72–72. DOI: 10.1109/MSPEC.2005.1491233.

[40] W. Ames. «Understanding spyware: risk and response». In: *IT Professional* 6.5 (2004), pp. 25–29. DOI: 10.1109/MITP.2004.71.

[41] Eric Chien. *Techniques of Adware and Spyware*. Tech. rep. Cupertino, CA, 2005.

[42] Jun Gao et al. «Should You Consider Adware as Malware in Your Study?» In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 604–608. DOI: 10.1109/SANER.2019.8668010.

[43] Monika, Pavol Zavarsky, and Dale Lindskog. «Experimental Analysis of Ransomware on Windows and Android Platforms: Evolution and Characterization». In: *Procedia Computer Science* 94 (2016). The 11th International Conference on Future Networks and Communications (FNC 2016) / The 13th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2016) / Affiliated Workshops, pp. 465–472. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.08.072. URL: https://www.sciencedirect.com/science/article/pii/S1877050916318221.

[44] Sana Aurangzeb et al. «Ransomware: A Survey and Trends». In: *Journal of Information Assurance and Security (ESCI - Thomson Reuters Indexed), ISSN: 1554-101* 12 (June 2017), pp. 48–.

[45] Harun Oz et al. «A Survey on Ransomware: Evolution, Taxonomy, and Defense Solutions». In: *ACM Comput. Surv.* 54.11s (Sept. 2022). ISSN: 0360-0300. DOI: 10.1145/3514229. URL: https://doi.org/10.1145/3514229.

[46] Craig Beaman et al. «Ransomware: Recent advances, analysis, challenges and future research directions». In: *Comput. Secur.* 111.C (Dec. 2021). ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102490. URL: https://doi.org/10.1016/j.cose.2021.102490.

[47] Sam L. Thomas and Aurélien Francillon. «Backdoors: Definition, Deniability and Detection». In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Michael Bailey et al. Cham: Springer International Publishing, 2018, pp. 92–113. ISBN: 978-3-030-00470-5.

[48] Fuad Mire Hassan Yahye Abukar Ahmed Mohd Aizaini Maarof and Mohamed Muse Abshir. «Survey of Keylogger Technologies». In: *Journal of Computer Science and Telecommunications* 5.2 (2014), pp. 25–31.

[49] Marco Salas-Nino et al. *The Evolution of Keylogger Technologies: A Survey from Historical Origins to Emerging Opportunities*. 2023. arXiv: 2312.10445 [cs.CR]. URL: https://arxiv.org/abs/2312.10445.

[50] Ekele Victoria C., Adebiyi Ayodele A., and Adebiyi Ayodele A. «Keylogger Detection: A Systematic Review». In: *2023 International Conference on Science, Engineering and Business for Sustainable Development Goals (SEB-SDG)*. Vol. 1. 2023, pp. 1–6. DOI: 10.1109/SEB-SDG57117.2023.10124477.

[51] Sérgio S.C. Silva et al. «Botnets: A survey». In: *Computer Networks* 57.2 (2013). Botnet Activity: Analysis, Detection and Shutdown, pp. 378–403. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2012.07.021. URL: https://www.sciencedirect.com/science/article/pii/S1389128612003568.

[52] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. «A Survey of Botnet and Botnet Detection». In: *2009 Third International Conference on Emerging Security Information, Systems and Technologies*. 2009, pp. 268–273. DOI: 10.1109/SECURWARE.2009.48.

[53] Meisam Eslahi, Rosli Salleh, and Nor Badrul Anuar. «Bots and botnets: An overview of characteristics, detection and challenges». In: *2012 IEEE International Conference on Control System, Computing and Engineering*. 2012, pp. 349–354. DOI: 10.1109/ICCSCE.2012.6487169.

[54] Sungkwan Kim et al. «A Brief Survey on Rootkit Techniques in Malicious Codes.» In: *J. Internet Serv. Inf. Secur.* 2.3/4 (2012), pp. 134–147.

[55] Jestin Joy, Anita John, and James Joy. «Rootkit Detection Mechanism: A Survey». In: *Advances in Parallel Distributed Computing*. Ed. by Dhinaharan Nagamalai, Eric Renault, and Murugan Dhanuskodi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 366–374. ISBN: 978-3-642-24037-9.

[56] Tyler Shields. «Survey of rootkit technologies and their impact on digital forensics». In: *Personal Communication* 11 (2008).

[57] Vinay M. Igure and Ronald D. Williams. «Taxonomies of attacks and vulnerabilities in computer systems». In: *IEEE Communications Surveys & Tutorials* 10.1 (2008), pp. 6–19. DOI: 10.1109/COMST.2008.4483667.

[58] Zhongqiang Chen, Yuan Zhang, and Zhongrong Chen. «A Categorization Framework for Common Computer Vulnerabilities and Exposures». In: *The Computer Journal* 53.5 (2010), pp. 551–580. DOI: 10.1093/comjnl/bxp040.

[59] Hazim Hanif et al. «The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches». In: *Journal of Network and Computer Applications* 179 (2021), p. 103009. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103009. URL: https://www.sciencedirect.com/science/article/pii/S1084804521000369.

[60] SecurityScorecard. *Vulnerabities by type*. Available at https://tinyurl.com/bdf4tr58 (2025/01/16).

[61] The MITRE Corporation. *The Common Weakness Enumeration*. Available at https://cwe.mitre.org/ (2025/01/16).

[62] William Patrick Arthur. «Control-Flow Security». PhD thesis. Michigan US: University of Michigan, 2016.

[63] Aleph One. «Smashing the Stack for Fun and Profit». In: *Phrack* 7.49 (1996).

[64] Donald Ray and Jay Ligatti. «Defining code-injection attacks». In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 179–190. ISSN: 0362-1340. DOI: 10.1145/2103621.2103678. URL: https://doi.org/10.1145/2103621.2103678.

[65] Marco Prandini and Marco Ramilli. «Return-Oriented Programming». In: *IEEE Security & Privacy* 10.6 (2012), pp. 84–87. DOI: 10.1109/MSP.2012.152.

[66] Tyler Bletsch et al. «Jump-oriented programming: a new class of code-reuse attack». In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: Association for Computing Machinery, 2011, pp. 30–40. ISBN: 9781450305648. DOI: 10.1145/1966913.1966919. URL: https://doi.org/10.1145/1966913.1966919.

[67] Shuo Chen et al. «Non-Control-Data Attacks Are Realistic Threats». In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005, p. 12.

[68] M Christodorescu et al. *Advances in Information Security: Malware detection*. 1st ed. New York, NY: Springer, 2007.

[69] J. Aycock. *Computer Viruses and Malware*. Advances in Information Security. Springer US, 2006. ISBN: 9780387341880.

[70] Ilsun You and Kangbin Yim. «Malware Obfuscation Techniques: A Brief Survey». In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 2010, pp. 297–300. DOI: 10.1109/BWCCA.2010.85.

[71] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. «Camouflage In Malware: From Encryption To Metamorphism». In: *International Journal of Computer Science And Network Security (IJCSNS)* 12 (Jan. 2012), pp. 74–83.

[72] Salvatore J. Stolfo, Ke Wang, and Wei-Jen Li. «Towards Stealthy Malware Detection». In: *Malware Detection*. Ed. by Mihai Christodorescu et al. Boston, MA: Springer US, 2007, pp. 231–249. ISBN: 978-0-387-44599-1.

[73] Ethan M. Rudd et al. «A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions». In: *IEEE Communications Surveys & Tutorials* 19.2 (2017), pp. 1145–1172. DOI: 10.1109/COMST.2016.2636078.

[74] Arini Balakrishnan and Chloe Schulze. «Code obfuscation literature survey». In: *CS701 Construction of compilers* 19 (2005), p. 31.

[75] Nor Zakiah Gorment, Ali Selamat, and Ondrej Krejcar. «Anti-Obfuscation Techniques: Recent Analysis of Malware Detection». In: *New Trends in Intelligent Software Methodologies, Tools and Techniques*. Ed. by Hamido Fujita, Yutaka Watanobe, and Takuya Azumi. Amsterdam: IOS Press, 2022, pp. 181–192.

[76] Hassan Jameel Asghar et al. «Use of cryptography in malware obfuscation». In: *Journal of Computer Virology and Hacking Techniques* 20.1 (2024), pp. 135–152.

[77] Wing Wong and Mark Stamp. «Hunting for metamorphic engines». In: *Journal in Computer Virology* 2 (Nov. 2006), pp. 211–229. DOI: 10.1007/s11416-006-0028-7.

[78] Evgenios Konstantinou. «Metamorphic Virus: Analysis and Detection». PhD thesis. London UK: University of London, 2008.

[79] Kenneth Brezinski, Ken Ferens, and Konstantinos Rantos. «Metamorphic Malware and Obfuscation: A Survey of Techniques, Variants, and Generation Kits». In: *Sec. and Commun. Netw.* 2023 (Sept. 2023). ISSN: 1939-0114. DOI: 10.1155/2023/8227751. URL: https://doi.org/10.1155/2023/8227751.

[80] *ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary*. 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[81] A. Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN: 9781999579517. URL: https://books.google.it/books?id=0jbxwQEACAAJ.

[82] Ian H. Witten et al. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128042915.

[83] Yanfang Ye et al. «A Survey on Malware Detection Using Data Mining Techniques». In: *ACM Computing Surveys* 50.3 (June 2017), pp. 1–40. ISSN: 1557-7341. DOI: 10.1145/3073559. URL: http://dx.doi.org/10.1145/3073559.

[84] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture Series. Cham, Switzerland: Springer Cham, 2020.

[85] N. Idika and P. Mathur. *A survey of malware detection techniques*. Tech. rep. West Lafayette, USA: Purdue University, 2007. URL: https://api.semanticscholar.org/CorpusID:2216347.

[86] Omer Aslan Aslan and Refik Samet. «A Comprehensive Review on Malware Detection Approaches». In: *IEEE Access* 8 (2020), pp. 6249–6271. DOI: 10.1109/ACCESS.2019.2963724.

[87] Khalid Alzarooni. «Malware variant detection». PhD thesis. London, U.K.: University College London, 2012.

[88] T. Sherwood et al. «Discovering and exploiting program phases». In: *IEEE Micro* 23.6 (2003), pp. 84–93. DOI: 10.1109/MM.2003.1261391.

[89] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. «Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management». In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 359–370. DOI: 10.1109/MICRO.2006.30.

[90] Corey Malone, Mohamed Zahran, and Ramesh Karri. «Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs». In: *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*. STC '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 71–76. ISBN: 9781450310017. DOI: 10.1145/2046582.2046596. URL: https://doi.org/10.1145/2046582.2046596.

[91] John Demme et al. «On the Feasibility of Online Malware Detection with Performance Counters». In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 559–570. ISSN: 0163-5964. DOI: 10.1145/2508148.2485970. URL: https://doi.org/10.1145/2508148.2485970.

[92] Hossein Sayadi et al. «Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification». In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465828.

[93] Hossein Sayadi et al. «2SMaRT: A Two-Stage Machine Learning-Based Approach for Run-Time Specialized Hardware-Assisted Malware Detection». In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019, pp. 728–733. DOI: 10.23919/DATE.2019.8715080.

[94] Nisarg Patel, Avesta Sasan, and Houman Homayoun. «Analyzing hardware based malware detectors». In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3062202.

[95] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. «Unsupervised Anomaly-Based Malware Detection Using Hardware Features». In: *Research in Attacks, Intrusions and Defenses*. Ed. by Angelos Stavrou, Herbert Bos, and Georgios Portokalidis. Cham: Springer International Publishing, 2014, pp. 109–129. ISBN: 978-3-319-11379-1.

[96] Hossein Sayadi et al. «Towards AI-Enabled Hardware Security: Challenges and Opportunities». In: *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2022, pp. 1–10. DOI: 10.1109/IOLTS56730.2022.9897507.

[97] Boyou Zhou et al. «Hardware Performance Counters Can Detect Malware: Myth or Fact?» In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ASIACCS '18. Incheon, Republic of Korea: Association for Computing Machinery, 2018, pp. 457–468. ISBN: 9781450355766. DOI: 10.1145/3196494.3196515. URL: https://doi.org/10.1145/3196494.3196515.

[98] Boyou Zhou et al. «A Cautionary Tale About Detecting Malware Using Hardware Performance Counters and Machine Learning». In: *IEEE Design & Test* 38.3 (2021), pp. 39–50. DOI: 10.1109/MDAT.2021.3063338.

[99] Marcus Botacin and André Grégio. «Why We Need a Theory of Maliciousness: Hardware Performance Counters in Security». In: *Information Security*. Ed. by Willy Susilo et al. Cham: Springer International Publishing, 2022, pp. 381–389. ISBN: 978-3-031-22390-7.

[100] Yoongu Kim et al. «Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors». In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964. DOI: 10.1145/2678373.2665726. URL: https://doi.org/10.1145/2678373.2665726.

[101] Onur Mutlu and Jeremie S. Kim. «RowHammer: A Retrospective». In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 39.8 (Aug. 2020), pp. 1555–1571. ISSN: 0278-0070. DOI: 10.1109/TCAD.2019.2915318. URL: https://doi.org/10.1109/TCAD.2019.2915318.

[102] Congmiao Li and Jean-Luc Gaudiot. «Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters». In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2019, pp. 588–597. DOI: 10.1109/COMPSAC.2019.00090.

[103] Xueyang Wang and Jerry Backer. *SIGDROP: Signature-based ROP Detection using Hardware Performance Counters*. 2016. arXiv: 1609.02667 [cs.CR].

[104] NIST. *National Vulnerability Database: CVE-2016-5195 Detail*. Available at http://https://nvd.nist.gov/vuln/detail/cve-2016-5195 (2024/31/01).

[105] Vincent M. Weaver and Sally A. McKee. «Can hardware performance counters be trusted?» In: *2008 IEEE International Symposium on Workload Characterization*. 2008, pp. 141–150. DOI: 10.1109/IISWC.2008.4636099.

[106] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. «Accuracy of performance counter measurements». In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 2009, pp. 23–32. DOI: 10.1109/ISPASS.2009.4919635.

[107] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. «Non-determinism and overcount on modern hardware performance counter implementations». In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2013, pp. 215–224. DOI: 10.1109/ISPASS.2013.6557172.

[108] Sanjeev Das et al. «SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security». In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 20–38. DOI: 10.1109/SP.2019.00021.

[109] Javier Barrera et al. «On the Reliability of Hardware Event Monitors in MPSoCs for Critical Domains». In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic: Association for Computing Machinery, 2020, pp. 580–589. ISBN: 9781450368667. DOI: 10.1145/3341105.3373955. URL: https://doi.org/10.1145/3341105.3373955.

[110] Sai Praveen Kadiyala et al. «Hardware Performance Counter-Based Fine-Grained Malware Detection». In: *ACM Trans. Embed. Comput. Syst.* 19.5 (Sept. 2020). ISSN: 1539-9087. DOI: 10.1145/3403943. URL: https://doi.org/10.1145/3403943.

[111] Marcus Ritter et al. «Conquering Noise With Hardware Counters on HPC Systems». In: *2022 IEEE/ACM Workshop on Programming and Performance Visualization Tools (ProTools)*. 2022, pp. 1–10. DOI: https://doi.org/pjbh.

[112] Muhammad Aditya Sasongko et al. «Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison». In: *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1594–1608. DOI: 10.1109/TPDS.2023.3257105.

[113] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. 767375. Intel. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[114] Ramon Canal et al. «VITAMIN-V: Virtual Environment and Tool-Boxing for Trustworthy Development of RISC-V Based Cloud Services». In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. 2023, pp. 302–308. DOI: 10.1109/DSD60849.2023.00050.

[115] Samuel Greengard. «Will RISC-V revolutionize computing?» In: *Commun. ACM* 63.5 (Apr. 2020), pp. 30–32. ISSN: 0001-0782. DOI: 10.1145/3386377. URL: https://doi.org/10.1145/3386377.

[116] B. Sprunt. «The basics of performance-monitoring hardware». In: *IEEE Micro* 22.4 (2002), pp. 64–71. DOI: 10.1109/MM.2002.1028477.

[117] Simone Dutto, Alessandro Savino, and Stefano Di Carlo. «Exploring Deep Learning for In-Field Fault Detection in Microprocessors». In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1456–1459. DOI: 10.23919/DATE51398.2021.9474120.

[118] Deniz Kasap et al. «Micro-Architectural features as soft-error markers in embedded safety-critical systems: preliminary study». In: *2023 IEEE European Test Symposium (ETS)*. 2023, pp. 1–5. DOI: 10.1109/ETS56758.2023.10174219.

[119] Alberto Carelli, Alessandro Vallero, and Stefano Di Carlo. «Shielding Performance Monitor Counters: a double edged weapon for safety and security». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 269–274. DOI: 10.1109/IOLTS.2018.8474191.

[120] Alberto Carelli, Alessandro Vallero, and Stefano Di Carlo. «Performance Monitor Counters: Interplay Between Safety and Security in Complex Cyber-Physical Systems». In: *IEEE Transactions on Device and Materials Reliability* 19.1 (2019), pp. 73–83. DOI: 10.1109/TDMR.2019.2898882.

[121] Mark Aldham et al. «Low-cost hardware profiling of run-time and energy in FPGA embedded processors». In: *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*. 2011, pp. 61–68. DOI: 10.1109/ASAP.2011.6043237.

[122] Ajay Nair, Karthik Shankar, and Roman Lysecky. «Efficient hardware-based nonintrusive dynamic application profiling». In: *ACM Trans. Embed. Comput. Syst.* 10.3 (May 2011). ISSN: 1539-9087. DOI: 10.1145/1952522.1952525. URL: https://doi.org/10.1145/1952522.1952525.

[123] Nam Ho, Paul Kaufmann, and Marco Platzner. «A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms». In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–4. DOI: 10.1109/FPL.2014.6927437.

[124] Enric Gibert et al. «Profiling Support for Runtime Managed Code: Next Generation Performance Monitoring Units». In: *IEEE Computer Architecture Letters* 14.1 (2015), pp. 62–65. DOI: 10.1109/LCA.2014.2321398.

[125] John L. Hennessy David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. San Francisco CA: Morgan Kaufmann, 2013.

[126] William Stallings. *Computer organization and architecture: designing for performance*. 10th ed. Hoboken NJ: Pearson Education, 2015.

[127] Sarah Harris David Harris. *Digital Design and Computer Architecture*. 2nd ed. San Francisco CA: Morgan Kaufmann, 2012.

[128] Chopra Rajiv. *Advanced Computer Architecture: A Practical Approach*. 1st ed. New Delhi: S Chand & Company, 2010.

[129] G.S. Sohi. «Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers». In: *IEEE Transactions on Computers* 39.3 (1990), pp. 349–359. DOI: 10.1109/12.48865.

[130] Denis Bakhvalov. *Performance Analysis and Tuning on Modern CPUs*. 2nd ed. Florida US: Independently published, 2024.

[131] Intel. *Perfmon Events*. Available at https://perfmon-events.intel.com (2024/12/12).

[132] Intel. *Perfmon Events*. Available at https://github.com/intel/perfmon (2024/12/12).

[133] *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*. 24593. Rev. 3.42. AMD. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

[134] *Performance Monitor Counters for AMD Family 1Ah Model 00h-0Fh Processors*. 58550. Rev. 0.02. AMD. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/58550-0.01.pdf.

[135] *Arm Architecture Reference Manual for A-profile architecture*. ARM DDI 0487. Rev. L.a. ARM. 2024. URL: https://tinyurl.com/488aymm9.

[136] RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Available at https://riscv.org (2025/01/08).

[137] Isabelle Guyon and André Elisseeff. «An introduction to variable and feature selection». In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1157–1182. ISSN: 1532-4435.

[138] Hudhaifa Mohammed Abdulwahab, S. Ajitha, and Mufeed Ahmed Naji Saif. «Feature Selection Techniques in the Context of Big Data: Taxonomy and Analysis». In: *Applied Intelligence* 52.12 (Sept. 2022), pp. 13568–13613. ISSN: 0924-669X. DOI: 10.1007/s10489-021-03118-3. URL: https://doi.org/10.1007/s10489-021-03118-3.

[139] Jundong Li et al. «Feature Selection: A Data Perspective». In: *ACM Comput. Surv.* 50.6 (Dec. 2017). ISSN: 0360-0300. DOI: 10.1145/3136625. URL: https://doi.org/10.1145/3136625.

[140] Girish Chandrashekar and Ferat Sahin. «A Survey on Feature Selection Methods». In: *Comput. Electr. Eng.* 40.1 (Jan. 2014), pp. 16–28. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2013.11.024. URL: https://doi.org/10.1016/j.compeleceng.2013.11.024.

[141] Shirley Browne et al. «PAPI: A Portable Interface to Hardware Performance Counters». In: *Proceedings of Department of Defense HPCMP Users Group Conference*. June 1999.

[142] Ingo Molnar and Thomas Gleixner. *Performance counters for Linux*. Available at https://https://lwn.net/Articles/337493 (2023/07/05).

[143] Nathan Binkert et al. «The gem5 simulator». In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[144] Nazareno Bruschi et al. «GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors». In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071.

[145] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[146] Yifeng Gao et al. «Adaptive-HMD: Accurate and Cost-Efficient Machine Learning-Driven Malware Detection using Microarchitectural Events». In: *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2021, pp. 1–7. DOI: 10.1109/IOLTS52814.2021.9486701.

[147] Hossein Sayadi et al. «Towards Accurate Run-Time Hardware-Assisted Stealthy Malware Detection: A Lightweight, yet Effective Time Series CNN-Based Approach». In: *Cryptography* 5.4 (2021). ISSN: 2410-387X. DOI: 10.3390/cryptography5040028. URL: https://www.mdpi.com/2410-387X/5/4/28.

[148] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. USA: Wiley-Interscience, 2000. ISBN: 0471056693.

[149] Gildo Torres and Chen Liu. «Where's Waldo? Identifying Anomalous Behavior of Data-Only Attacks Using Hardware Features». In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF '22. Turin, Italy: Association for Computing Machinery, 2022, pp. 75–84. ISBN: 9781450393386. DOI: 10.1145/3528416.3530226. URL: https://doi.org/10.1145/3528416.3530226.

[150] Karl Pearson. «Note on regression and inheritance in the case of two parents». In: *Proc. R. Soc. Lond.* 58 (1895), pp. 240–242. DOI: 10.1098/rspl.1895.0041.

[151] Hanchuan Peng, Fuhui Long, and C. Ding. «Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.8 (2005), pp. 1226–1238. DOI: 10.1109/TPAMI.2005.159.

[152] Baljit Singh et al. «On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters». In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 483–493. ISBN: 9781450349444. DOI: 10.1145/3052973.3052999. URL: https://doi.org/10.1145/3052973.3052999.

[153] Abigail Kwan. *Malware Detection at the Microarchitecture Level using Machine Learning Techniques.* Available at http://https://scholarworks.calstate.edu/downloads/mk61rp641 (2024/31/01).

[154] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[155] E. Frank et al. «Weka: A machine learning workbench for data mining.» In: *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*. Ed. by O. Maimon and L. Rokach. Berlin: Springer, 2005, pp. 1305–1314. URL: http://researchcommons.waikato.ac.nz/handle/10289/1497.

[156] Khaled N. Khasawneh et al. «Ensemble Learning for Low-Level Hardware-Supported Malware Detection». In: *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*. RAID 2015. Kyoto, Japan: Springer-Verlag, 2015, pp. 3–25. ISBN: 9783319263618. DOI: 10.1007/978-3-319-26362-5_1. URL: https://doi.org/10.1007/978-3-319-26362-5%5C_1.

[157] Hossein Sayadi et al. «Machine Learning-Based Approaches for Energy-Efficiency Prediction and Scheduling in Composite Cores Architectures». In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017, pp. 129–136. DOI: 10.1109/ICCD.2017.28.

[158] Charalambos Konstantinou et al. «HPC-Based Malware Detectors Actually Work: Transition to Practice After a Decade of Research». In: *IEEE Design & Test* 39.4 (2022), pp. 23–32. DOI: 10.1109/MDAT.2022.3143438.

[159] DARPA. *RADICS: Rapid Attack Detection, Isolation and Characterization Systems.* Available at https://https://www.darpa.mil/research/programs/rapid-attack-detection (2025/02/13).

[160] Intel. *Detect Ransomware and Other Advanced Threats with Intel Threat Detection Technology.* Available at https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/hardware-shield/threat-detection-technology/detect-ransomware-solution-brief.html (2023/10/02).

[161] Microsoft. *Defending against cryptojacking with Microsoft Defender for Endpoint and Intel TDT.* Available at https://www.microsoft.com/en-us/security/blog/2021/04/26/defending-against-cryptojacking-with-microsoft-defender-for-endpoint-and-intel-tdt (2023/10/02).

[162] Meltem Ozsoy et al. «Malware-aware processors: A framework for efficient online malware detection». In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 651–661. DOI: 10.1109/HPCA.2015.7056070.

[163] Chronicle. *VirusTotal*. Available at `https://www.virustotal.com/gui/home/upload` (2023/06/29).

[164] Yanchen Qiao, Xiaochun Yun, and Yongzheng Zhang. «How to Automatically Identify the Homology of Different Malware». In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 929–936. DOI: `10.1109/TrustCom.2016.0158`.

[165] M.R. Guthaus et al. «MiBench: A free, commercially representative embedded benchmark suite». In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: `10.1109/WWC.2001.990739`.

[166] Thomas G. Dietterich. «Ensemble Methods in Machine Learning». In: *Multiple Classifier Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15. ISBN: 978-3-540-45014-6.

[167] Yoav Freund and Robert E Schapire. «A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting». In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000. DOI: `https://doi.org/10.1006/jcss.1997.1504`. URL: `https://www.sciencedirect.com/science/article/pii/S002200009791504X`.

[168] Leo Breiman. «Bagging predictors». In: *Machine Learning* 24.2 (1996), pp. 123–140.

[169] Zhiguang Wang, Weizhong Yan, and Tim Oates. «Time series classification from scratch with deep neural networks: A strong baseline». In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 1578–1585. DOI: `10.1109/IJCNN.2017.7966039`.

[170] Fazle Karim et al. «LSTM Fully Convolutional Networks for Time Series Classification». In: *IEEE Access* 6 (2018), pp. 1662–1669. DOI: `10.1109/ACCESS.2017.2779939`.

[171] Patrick Schäfer. «Scalable time series classification». In: *Data Mining and Knowledge Discovery* 30.5 (Sept. 2016), pp. 1273–1298. ISSN: 1573-756X. DOI: `10.1007/s10618-015-0441-y`. URL: `https://doi.org/10.1007/s10618-015-0441-y`.

[172] Xiaosheng Li and Jessica Lin. «Linear Time Complexity Time Series Classification with Bag-of-Pattern-Features». In: *2017 IEEE International Conference on Data Mining (ICDM)*. 2017, pp. 277–286. DOI: `10.1109/ICDM.2017.37`.

[173] Hossein Sayadi et al. «StealthMiner: Specialized Time Series Machine Learning for Run-Time Stealthy Malware Detection based on Microarchitectural Features». In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. GLSVLSI '20. Virtual Event, China: Association for Computing Machinery, 2020, pp. 175–180. ISBN: 9781450379441. DOI: 10.1145/3386263.3407585. URL: https://doi.org/10.1145/3386263.3407585.

[174] Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo. «Zero-Day Hardware-Supported Malware Detection of Stack Buffer Overflow Attacks: An Application Exploiting the CV32e40p RISC-V Core». In: *2025 IEEE 26th Latin American Test Symposium (LATS)*. 2025, pp. 1–6. DOI: 10.1109/LATS65346.2025.10963939.

[175] Scikit-learn project. *Novelty and Outlier Detection*. Available at https://scikit-learn.org/stable/modules/outlier_detection.html (2025/02/15).

[176] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020.

[177] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[178] Markus M. Breunig et al. «LOF: identifying density-based local outliers». In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: Association for Computing Machinery, 2000, pp. 93–104. ISBN: 1581132174. DOI: 10.1145/342009.335388. URL: https://doi.org/10.1145/342009.335388.

[179] Alberto Garcia-Serrano. *Anomaly Detection for malware identification using Hardware Performance Counters*. 2015. arXiv: 1508.07482 [cs.CR].

[180] Shachar Siboni and Asaf Cohen. *Universal Anomaly Detection: Algorithms and Applications*. 2015. arXiv: 1508.03687 [cs.CR].

[181] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. «Isolation Forest». In: *2008 Eighth IEEE International Conference on Data Mining*. 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.

[182] Peter J. Rousseeuw and Katrien Van Driessen. «A Fast Algorithm for the Minimum Covariance Determinant Estimator». In: *Technometrics* 41.3 (1999), pp. 212–223. DOI: 10.1080/00401706.1999.10485670. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1999.10485670. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1999.10485670.

[183] Michael Gautschi et al. «Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[184] Tao Liu, Zhen Zhou, and Lijun Yang. «Layered isolation forest: A multi-level subspace algorithm for improving isolation forest». In: *Neurocomputing* 581 (2024), p. 127525. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2024.127525. URL: https://www.sciencedirect.com/science/article/pii/S0925231224002960.

[185] Bruno Davide. «Malware detection using hardware performance counters on RISC-V based cloud servers». MA thesis. Turin, Italy: Politecnico di Torino, Oct. 2024.

[186] Av-Test Institute. *Av-Atlas Malware Statistics*. Available at https://portal.av-atlas.org/malware/statistics (2025/08/04).

[187] Emanuele Cozzi et al. «Understanding Linux Malware». In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 161–175. DOI: 10.1109/SP.2018.00054.

[188] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR]. URL: https://arxiv.org/abs/2007.03152.

[189] Peter Yuen Ho Hin et al. «Supporting RISC-V full system simulation in gem5». In: *Proc. Workshop Comput. Architect. Res. RISC-V*. 2021.

[190] Alec Roelke and Mircea R Stan. «Risc5: Implementing the RISC-V ISA in gem5». In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. Vol. 7. 17. 2017.

[191] Tuan Ta, Lin Cheng, and Christopher Batten. «Simulating multi-core RISC-V systems in gem5». In: *Workshop on Computer Architecture Research with RISC-V*. 2018.

[192] FFmpeg Community. *FFmpeg Project*. Available at https://www.ffmpeg.org/ (2025/27/03).

[193] Restic Community. *Restic Project*. Available at https://restic.net/ (2025/27/03).

[194] Arch Linux Developers. *Ag Manual*. Available at https://man.archlinux.org/man/extra/the_silver_searcher/ag.1.en (2025/27/03).

[195] Petr Cech. *Unrar - Linux Manual Page*. Available at https://linux.die.net/man/1/unrar (2025/27/03).

[196] Gordon Lyon. *Nmap Project*. Available at https://nmap.org/ (2025/27/03).

[197] Luis MartinGarcia and Gordon Lyon. *Nping*. Available at https://nmap.org/nping/ (2025/27/03).

[198] Tarcisio Marinho. *GonnaCry Repository*. Available at https://github.com/tarcisio-marinho/GonnaCry (2025/27/03).

[199] Md Omar Faruk Rokon et al. «SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub». In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 149–163. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/omar.

[200] Smelly. *Vx-underground*. Available at https://vx-underground.org/ (2025/27/03).

[201] Dev0uss. *Gcc-DDOS-Attacks Repository*. Available at https://github.com/Dev0uss/Gcc-DDOS-Attacks/tree/master (2025/27/03).

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.